Distributed Subgraph Matching on Timely Dataflow [Experiments and Analyses]

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[♭]

[§] The University of New South Wales, Sydney, Australia
[♮]East China Normal University, China
[‡]Centre for Artificial Intelligence, University of Technology, Sydney, Australia
[♭] Alibaba Group, China

[§]{llai,zyang,zhangw,lxue, khao}@cse.unsw.edu.au; [♮]{zhuqing,xinjin,zmlai,rwang}@stu.ecnu.edu.cn;
[‡]{lu.qin,ying.zhang}@uts.edu.au; [♭]{zhengping.qzp, jingren.zhou}@alibaba-inc.com

ABSTRACT

Recently there emerge many distributed algorithms that aim at solving subgraph matching at scale. Existing algorithm-level comparisons failed to provide a systematic view to the pros and cons of each algorithm mainly due to the intertwining of strategy and optimization. In this paper, we identify four strategies and three general-purpose optimizations from representative state-of-the-art works. We implement the four strategies with the optimizations based on the common `Timely` dataflow system for systematic strategy-level comparison. Our implementation covers all representation algorithms. We conduct extensive experiments for both unlabelled matching and labelled matching to analyze the performance of distributed subgraph matching under various settings, which is finally summarized as a practical guide.

PVLDB Reference Format:
Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Wenjie Zhang, Ying Zhang, Xuemin Lin, Lu Qin, Zhengping Qian and Jingren Zhou. Distributed Subgraph Matching on Timely Dataflow [Experiments and Analyses]. *PVLDB*, xx(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/TBD

1. INTRODUCTION

Given a query graph Q and a data graph G, subgraph matching is defined as finding all subgraph instances of G that are isomorphic to Q. In this paper, we assume that the query graph and data graph are undirected [1] simple graphs, and may be unlabelled or labelled. In this work, we mainly focus on unlabelled case given that most distributed algorithms are developed under this setting. We also demonstrate some results of labelled case due to its practical usefulness. Subgraph matching is one of the most fundamental operations in graph analysis, and has been used in a wide spectrum of applications [28, 17, 38].

As subgraph matching problem is in general computationally intractable [46], and data graph nowadays is growing beyond the capacity of one single machine, people are seeking efficient and scalable algorithms in the distributed context. Unless otherwise specified, in this paper we consider a simple **hash partition** of the graph data, that is the graph is randomly partitioned by vertices, and each vertex's neighbors will be placed in the same partition.

By treating query vertices as attributes and the matched results as relational tables, we can express subgraph matching via natural joins. The problem is accordingly transformed into seeking optimal join plan, where the optimization goal is typically to minimize the communication cost. In this paper, we focus on an in-depth survey and comparison of representative distributed subgraph matching algorithms that follow such join scheme.

1.1 State-of-the-arts.

In order to solve subgraph matching using join, existing works studied various join strategies, which can be categorized into three classes, namely "Binary-join-based subgraph-growing algorithms" (BinJoin), "Worst-case optimal vertex-growing algorithms" (WOptJoin) and "Shares of Hypercube" (ShrCube). We also include Others category for algorithms that do not clearly belong to the above categories.

BinJoin. The strategy computes subgraph matching via a series of binary joins. It first decomposes the original query graph into a set of *join units* whose matches can serve the base relation of the join. Then the base relations are joined based on a predefined *join order*. The BinJoin algorithms differ in the selections of join unit and join order. Typical choices of join unit are star (a tree of depth 1) in StarJoin [50], `TwinTwig` (an edge or intersection of two edges) in TwinTwigJoin [36], and clique (a graph whose vertices are mutually connected) in CliqueJoin [38]. Most existing algorithms adopt the easier-solving left-deep join order [32] except CliqueJoin, which explores the optimality-guaranteed bushy join [32].

WOptJoin. Given $\{v_0, v_1, \cdots, v_n\}$ as the query vertices, WOptJoin strategy first computes all matches of $\{v_0\}$ that can present in the results, then matches of $\{v_0, v_1\}$, and so

[1]Our implementation can seamlessly handle directed case.

Proceedings of the VLDB Endowment, Vol. xx, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/TBD

forth until constructing the results. Ngo et al. proposed the worst-case optimal join algorithm `GenericJoin` [43], based on which Ammar et al. implemented BiGJoin in `Timely` dataflow system [42] and showed its worst-case optimality [14]. In this paper, we also find out that the BinJoin algorithm CliqueJoin (with "overlapped decomposition" [2]) is also a variant of `GenericJoin`, and is hence worst-case optimal.

ShrCube. ShrCube strategy treats the computation of the query with n vertices as an n-dimensional *hypercube*. It partitions the hypercube across w workers in the cluster, and then each worker can compute its own share locally with no need of exchanging data. As a result, it typically renders much less communication cost than that of BinJoin and WOptJoin algorithms. MultiwayJoin adopts the idea of ShrCube for subgraph matching. In order to properly partition the computation without missing results, MultiwayJoin needs to duplicate each edge in multiple workers. As a result, MultiwayJoin can almost carry the whole graph in each worker for certain queries [36, 14] and thus scale out poorly.

Others. Shao et al. proposed PSgL [49] that processes subgraph matching via breadth-first-style traversal. Staring from an initial query vertex, PSgL iteratively expands the partial results by merging the matches of certain vertex's unmatched neighbors. It has been pointed out in [36] that PSgL is actually a variant of StarJoin. Very recently, Qiao et al. proposed CrystalJoin [44] that aims at resolving the "output crisis" by compressing the (intermediate) results. The idea is to first compute the matches of the vertex cover of the query graph, then the remaining vertices' matches can be compressed as intersection of the vertex cover's neighbors to avoid costly cartesian product.

Optimizations. Apart from join strategies, existing algorithms also explored a variety of optimizations, some of which are query- or algorithm-specific, while we spotlight three general-purpose optimizations, `Batching`, `TrIndexing` and `Compression`. `Batching` aims to divide the whole computation into sub-tasks that can be evaluated independently in order to save resource (memory) allocation. `TrIndexing` precomputes and indices the triangles (3-cycles) of the graph to facilitate pruning. `Compression` attempts to maintain the (intermediate) results in a compressed form to reduce resource allocation and communication cost.

1.2 Motivations.

In this paper, we survey seven representative algorithms to solve distributed subgraph matching: StarJoin [50], MultiwayJoin [13], PSgL [49], TwinTwigJoin [36], CliqueJoin [38], CrystalJoin [44] and BiGJoin [14]. While all these algorithms embody some good merits in theory, existing **algorithm-level** comparisons failed to provide a systematic view to the pros and cons of each algorithm due to several reasons. Firstly, the prior experiments did not take into consideration the differences of languages and the cost of the systems on which each implementation is based (Table 1). Secondly, some implementations hardcode query-specific optimizations for each query, which makes it hard to judge whether the observed performance is from the algorithmic advancement or hardcoded optimization. Thirdly, all BinJoin and WOptJoin algorithms (more precisely, their implementations) intertwined join strategy with some optimizations of `Batching`, `TrIndexing` and `Compression`. We show in Table 1 how each optimization has been applied in current implementation. For example, CliqueJoin only adopted `TrIndexing` and some query-specific `Compression`, while BiGJoin considered `Batching` in general, but `TrIndexing` only for one specific query (`Compression` was only discussed in paper, but not implemented). People naturally wonder that *"maybe it is better to adopt A strategy with B optimization"*, but unfortunately none of existing implementation covers that combination. Last but not least, there misses an important benchmarking of the FullRep strategy, that is to maintain the whole graph in each partition and parallelize embarrassingly [30]. FullRep strategy requires no communication, and it should be the most efficient strategy when each machine can hold the whole graph (the case for most experimental settings nowadays).

Table 1 summarizes the surveyed algorithms via the category of strategy, the optimality guarantee, and the status of current implementations including the based platform and how the three optimizations are adopted.

1.3 Our Contributions

To address the above issues, we aims at a systematic, **strategy-level** benchmarking of distributed subgraph matching in this paper. To achieve that goal, we implement all strategies, together with the three general-purpose optimizations for subgraph matching based on the `Timely` dataflow system [42]. Note that our implementation covers all seven representative algorithms. Here, we use `Timely` as the base system as it incurs less cost [41] than other popular systems like Giraph [5], Spark [53] and GraphLab [39], so that the system's impact can be reduced to the minimum.

We implement the benchmarking platform using our best effort based on the papers of each algorithm and email communications with the authors. Our implementation is (1) **generic** to handle arbitrary query, and does not include any hardcoded optimizations; (2) **flexible** that can configure `Batching`, `TrIndexing` and `Compression` optimizations in any combination for BinJoin and WOptJoin algorithms; and (3) **efficient** that are comparable to and sometimes even faster than the original hardcoded implementation. Note that the three general-purpose optimizations are mainly used to reduce communication cost, and is not useful to the ShrCube and FullRep strategies, while we still devote a lot of efforts into their implementations. Aware that their performance heavily depends on the local algorithm, we implement and compare the state-of-the-art local subgraph matching algorithms proposed in [35], [12] (for unlabelled matching), and [17] (for labelled matching), and adopt the best-possible implementation. For ShrCube, we refer to [21] to implement "Hypercube Optimization" for better hypercube sharing.

We make the following contributions in the paper.

(1) A benchmarking platform based on `Timely` dataflow system for distributed subgraph matching. We implement four distributed subgraph matching strategies (and the general optimizations) that covers seven state-of-the-art algorithms: StarJoin [50], MultiwayJoin [13], PSgL [49], TwinTwigJoin [36], CliqueJoin [38], CrystalJoin [44] and BiGJoin [14]. Our implementation is generic to handle arbitrary query, including the labelled and directed query, and thus can guide practical use.

(2) Three general-purpose optimizations - `Batching`, `TrIndexing` and `Compression`. We investigate the literature on the optimization strategies, and spotlight the

[2] Decompose the query graph into join units that are allowed to overlap edges

Algorithm	Category	Worst-case Optimality	Platform	Optimizations
StarJoin [50]	BinJoin	No	Trinity [48]	None
MultiwayJoin [13]	ShrCube	N/A	Hadoop [36], Myria [21]	N/A
PSgL [49]	Others	No	Giraph [5]	None
TwinTwigJoin [36]	BinJoin	No	Hadoop	`Compression` [37]
CliqueJoin [38]	BinJoin	Yes (Section 6)	Hadoop	`TrIndexing`, some `Compression`
CrystalJoin [44]	Others	N/A	Hadoop	`TrIndexing`, `Compression`
BiGJoin [14]	WOptJoin	Yes [14]	Timely Dataflow [42]	`Batching`, specific `TrIndexing`

Table 1: Summarization of the surveyed algorithms.

three general-purpose optimizations. We propose heuristics to incorporate the three optimizations into BinJoin and WOptJoin strategies, with no need of query-specific adjustments from human experts. The three optimizations can be flexibly configured in any combination.

(3) In-depth experimental studies. In order to extensively evaluate the performance of each strategy and the effectiveness of the optimizations, we use data graphs of different sizes and densities, including sparse road network, dense ego network, and web-scale graph that is larger than each machine's configured memory. We select query graphs of various characteristics that are either from existing works or suitable for benchmarking purpose. In addition to running time, we measure the communication cost, memory usage and other metrics to help reason the performance.

(4) A practical guide of distributed subgraph matching. Through empirical analysis covering the variances of join strategies, optimizations, join plans, we propose a practical guide for distributed subgraph matching. We also inspire interesting future work based on the experimental findings.

1.4 Organizations

. The rest of the paper is organized as follows. Section 2 defines the problem of subgraph matching and introduces preliminary knowledge. Section 3 surveys the representative algorithms, and our implementation details following the categories of BinJoin, WOptJoin, ShrCube and Others. Section 4 investigates the three general-purpose optimizations and devises heuristics of applying them to BinJoin and WOptJoin algorithms. Section 5 demonstrates the experimental results and our in-depth analysis. Section 7 discusses the related works, and Section 8 concludes the whole paper.

2. PRELIMINARIES

2.1 Problem Definition

Graph Notations. A graph g is defined as a 3-tuple, $g = (V_g, E_g, L_g)$, where V_g is the vertex set and $E_g \subseteq V_g \times V_g$ is the edge set of g, and L_g is a label function that maps each vertex $\mu \in V_g$ and/or each edge $e \in E_g$ to a label. Note that for unlabelled graph, L_g simply maps all vertices and edges to $\emptyset$. For a vertex $\mu \in V_g$, denote $\mathcal{N}_g(\mu)$ as the set of neighbors, $d_g(\mu) = |\mathcal{N}_g(\mu)|$ as the degree of μ, $\overline{d}_g = \frac{2|E_g|}{|V_g|}$ and $D_g = \max_{\mu \in V(g)} d_g(\mu)$ as the average and maximum degree, respectively. A *subgraph* g' of g, denoted $g' \subseteq g$, is a graph that satisfies $V_{g'} \subseteq V_g$ and $E_{g'} \subseteq E_g$.

Given $V' \subseteq V_g$, we define induced subgraph $g(V')$ as the subgraph induced by V', that is $g(V') = (V', E(V'), L_g)$, where $E(V') = \{e = (\mu, \mu') \mid e \in E_g, \mu \in V' \wedge \mu' \in V'\}$. We say $V' \subseteq V_g$ is a vertex cover of g, if $\forall\ e = (\mu, \mu') \in E_g$, $\mu \in V'$ or $\mu' \in V'$. A minimum vertex cover V_g^c is a vertex cover of g that contains minimum number of vertices. A connected vertex cover is a vertex cover whose induced subgraph is connected, among which a minimum connected vertex cover, denoted V_g^{cc}, is the one with the minimum number of vertices.

Data and Query Graph. We denote the data graph as G, and let $N = |V_G|$, $M = |E_G|$. Denote a data vertex of id i as u_i where $1 <= i <= N$. Note that the data vertex has been reordered such that if $d_G(u) < d_G(u')$, then $id(u) < id(u')$. We denote the query graph as Q, and let $n = |V_Q|$, $m = |E_Q|$, and $V_Q = \{v_1, v_2, \cdots, v_n\}$.

Subgraph Matching. Given a data graph G and a query graph Q, we define *subgraph isomorphism*:

Definition 1. (Subgraph Isomorphism.) *Subgraph isomorphism* is defined as a bijective mapping $f : V(Q) \to V(G)$ such that, (1) $\forall v \in V(Q)$, $L_Q(v) = L_G(f(v))$; (2) $\forall (v, v') \in E(Q)$, $(f(v), f(v')) \in E(G)$, and $L_Q((v, v')) = L_G((f(v), f(v')))$. A subgraph isomorphism is called a *Match* in this paper. With the query vertices listed as $\{v_1, v_2, \cdots, v_n\}$, we can simply represent a match f as $\{u_{k_1}, u_{k_2}, \cdots, u_{k_n}\}$, where $f(v_i) = u_{k_i}$ for $1 <= i <= n$.

The *Subgraph Matching* problem aims at finding all matches of Q in G. Denote $R_G(Q)$ (or $R(Q)$ when the context is clear) as the result set of Q in G. As prior works [36, 38, 49], we apply *symmetry breaking* for unlabelled matching to avoid duplicate enumeration caused by automorphism. Specifically, we first assign partial order O_Q to the query graph according to [27]. Here, $O_Q \subseteq V_Q \times V_Q$, and $(v_i, v_j) \in O_Q$ means $v_i < v_j$. In unlabelled matching, a match f must satisfy the *order constraint*: $\forall (v, v') \in O_Q$, it holds $f(v) < f(v')$. Note that we do **not** consider order constraint in labelled matching.

Example 1. In Figure 1, we present a query graph Q and a data graph G. For unlabelled matching, we give the partial order O_Q under the query graph. There are three matches: $\{u_1, u_2, u_6, u_5\}$, $\{u_2, u_5, u_3, u_6\}$ and $\{u_4, u_3, u_6, u_5\}$. It is easy to check that these matches satisfy the order constraint. Without the order constraint, there are actually four automorphic[3] matches corresponding to each above match [13]. For labelled matching, we use different fillings to represent the labels. There are two matches accordingly - $\{u_1, u_2, u_6, u_5\}$ and $\{u_4, u_3, u_6, u_5\}$.

By treating the query vertices as attributes and data edges as relational table, we can write subgraph matching query

[3]Automorphism is an isomorphism from one graph to itself.

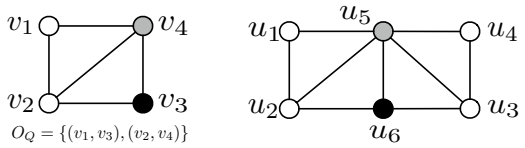


Figure 1: Query Graph Q (Left) and Data Graph G (Right).

(a) Left-deep join plan (b) Bushy join plan

Figure 2: Simple BinJoin Join Plans.

as a multiway-way join of the edge relations. For example, regardless of label and order constraints, the query of Example 1 can be written as the following join

$$R(Q) = E(v_1, v_2) \bowtie E(v_2, v_3) \bowtie E(v_3, v_4) \bowtie E(v_1, v_4) \bowtie E(v_2, v_4). \quad (1)$$

This motivates researchers to leverage join operation for large-scale subgraph matching, given that join can be easily distributed, and it is natively supported in many distributed data engines like Spark [53] and Flink [18].

2.2 Timely Dataflow System

`Timely` is a distributed data-parallel dataflow system [42]. The minimum processing unit of `Timely` is a *worker*, which can be simply seen as a process that occupies a CPU core. Typically, one physical multi-core machine can run several workers. `Timely` follows the *shared-nothing dataflow* computation model [23] that abstracts the computation as a dataflow graph. In the dataflow graph, the vertex (a.k.a. *operator*) defines the computing logics and the edges in between the operators represent the data streams. One operator can accept multiple input streams, feed them to the computing, and produce (typically) one output stream. After the dataflow graph for certain computing task is defined, it is distributed to each worker in the cluster, and further translated into a physical execution plan. Based on the physical plan, each worker can accordingly process the task in parallel while accepting the corresponding input portion.

3. ALGORITHM SURVEY

We survey the distributed subgraph matching algorithms following the categories of BinJoin, WOptJoin, ShrCube, and Others. We also show that CliqueJoin is a variant of `GenericJoin` [43], and is thus worst-case optimal.

3.1 BinJoin

The simplest BinJoin algorithm uses data edges as the base relation, which starts from one edge, and expands by one edge in each join. For example, to solve the join of Equation 1, a simple plan is shown in Figure 2a. The join plan is straightforward, but the intermediate results, especially R_2 (a 3-path), can be huge.

To improve the performance of BinJoin, people devoted their efforts into: (1) using more complex base relations other than edge; (2) devising better join plan P. The base relations $B_{[q]}$ represent the matches of a set of sub-structures $[q]$ of the query graph Q. Each $p \in [q]$ is called a join unit, and it must satisfy $V_Q = \bigcup_{p\in[q]} V_p$ and $E_Q = \bigcup_{p\in[q]} E_p$. With the data graph partitioned across the cluster, [38] constrains the join unit to be the structure whose results can be independently computed within each partition (i.e. embarrassingly parallel [30]). It is not hard to see that when each vertex has full access to the neighbors in the partition, we can compute the matches of a k-star (a star of k leaves) rooted on the vertex u by enumerating all k-combinations within $\mathcal{N}_G(u)$. Therefore, star is a qualified and indeed widely used join unit.

Given the base relations, the join plan P determines an order of processing binary joins. A join plan is *left-deep*[4] if there is at least a base relation involved in each join, otherwise it is *bushy*. For example, the join plan in Figure 2a is left-deep, and a bushy join plan is shown in Figure 2b. Note that the bushy plan avoids the expensive R_2 in the left-deep plan, and is generally better.

StarJoin. As the name suggests, StarJoin uses star as the join unit, and it follows the left-deep join order. To decompose the query graph, it first locates the vertex cover of the query graph, and each vertex in the cover and its unused neighbors naturally form a star [50]. A StarJoin plan for Equation 1 is

$$(J_1)\ \ R(Q) = \texttt{Star}(v_2; \{v_1, v_3, v_4\}) \bowtie \texttt{Star}(v_4; \{v_2, v_3\}),$$

where $\texttt{Star}(r; L)$ denotes a `Star` relation (the matches of the star) with r as the root, and L as the set of leaves.

TwinTwigJoin. Enumerating a k-star on a vertex of degree d will render $O(d^k)$ cost. We refer *star explosion* to the case while enumerating stars on a large-degree vertex. Lai et al. proposed TwinTwigJoin [36] to address the issue of StarJoin by forcing the join plan to use `TwinTwig` (a star of at most two edges) instead of a general star as the join unit. Intuitively, this would help ameliorate the star explosion by constraining the cost of each join unit from d^k of arbitrary k to at most d^2. TwinTwigJoin follows StarJoin to use left-deep join order. The authors proved that TwinTwigJoin is instance optimal to StarJoin, that is given any general StarJoin plan in the left-deep join order, we can rewrite it as an alternative TwinTwigJoin plan that draws no more cost (in the big O sense) than the original StarJoin, where the cost is evaluated based on Erdös-Rényi random graph (ER) model [24]. A TwinTwigJoin plan for Equation 1 is

$$\begin{aligned}(J_1)\ \ & R_1(v_1, v_2, v_3, v_4) = \\ & \texttt{TwinTwig}(v_1; \{v_2, v_4\}) \bowtie \texttt{TwinTwig}(v_2; \{v_3, v_4\}); \quad (2)\\ (J_2)\ \ & R(Q) = R_1(v_1, v_2, v_3, v_4) \bowtie \texttt{TwinTwig}(v3; \{v4\}),\end{aligned}$$

where $\texttt{TwinTwig}(r; L)$ denotes a `TwinTwig` relation with r as the root, and L as the leaves.

CliqueJoin. TwinTwigJoin hampers star explosion to some extent, but still suffers from the problems of long execution ($\Omega(\frac{m}{2})$ rounds) and suboptimal left-deep join plan. CliqueJoin resolves the issues by extending StarJoin in two aspects. Firstly, CliqueJoin applies the "triangle partition" strategy (Section 4.2), which enables CliqueJoin to use clique, in addition to star, as the join unit. The use of clique can greatly shorten the execution especially when the query is dense, although it still degenerates to StarJoin when the query contains no clique subgraph. Secondly, CliqueJoin exploits the bushy join plan to approach optimality. A CliqueJoin plan for Equation 1 is:

$$(J_1)\ \ R(Q) = \texttt{Clique}(\{v_1, v_2, v_4\}) \bowtie \texttt{Clique}(\{v_2, v_3, v_4\}), \quad (3)$$

where $\texttt{Clique}(V)$ denotes a `Clique` relation of the involving vertices V.

[4]More precisely it is deep, and can further be left-deep and right-deep. In this paper, we assume that it is left-deep following the prior work [36].

Implementation Details. We implement the BinJoin strategy based on the join framework proposed in [38] to cover StarJoin, TwinTwigJoin and CliqueJoin.

We use power-law random graph (PR) model [22] to estimate the cost as [38], and implement the dynamic programming algorithm [38] to compute the cost-optimal join plan. Once the join plan is computed, we translate the plan into `Timely` dataflow that processes each binary join using a `Join` operator. We implement the `Join` operator following `Timely`'s official "pipeline" `HashJoin` example[5]. We modify it into "batching-style" - the mappers (senders) shuffle the data based on the join key, while the reducers (receivers) maintain the received key-value pairs in a hash table (until mapper completes) for join processing. The reasons that we implement the join as "batching-style" are, (1) its performance is similar to "pipeline" join as a whole; (2) it replays the original implementation in Hadoop; and (3) it favors the `Batching` optimization (Section 4.1).

3.2 WOptJoin

WOptJoin strategy processes subgraph matching by matching vertices in a predefined order. Given the query graph Q and $V_Q = \{v_1, v_2, \cdots, v_n\}$ as the matching order, the algorithm starts from an empty set, and computes the matches of the subset $\{v_1, \cdots, v_i\}$ in the i^{th} rounds. Denote the partial results after the i^{th} $(i < n)$ round as R_i, and $p = \{u_{k_1}, u_{k_2}, \cdots, u_{k_i}\} \in R_i$ is one of the tuples. In the $i + 1^{th}$ round, the algorithm expands the results by matching v_{i+1} with $u_{k_{i+1}}$ for p iff. $\forall_{1 \le j \le i}(v_j, v_{i+1}) \in E_Q$, $(u_{k_j}, u_{k_{i+1}}) \in E_G$. It is immediate that the candidate matches of v_{i+1}, denoted $C(v_{i+1})$, can be obtained by intersecting the relevant neighbors of the matched vertices as

$$C(v_{i+1}) = \bigcap_{\forall_{1 \le j \le i} \wedge (v_j, v_{i+1}) \in E_Q} \mathcal{N}_G(u_{k_j}). \quad (4)$$

BiGJoin. BiGJoin adopts the WOptJoin strategy in `Timely` dataflow system. The main challenge is to implement the intersection efficiently using `Timely` dataflow. For that purpose, the authors designed the following three operators:

- `Count`: Checking the number of neighbors of each u_{k_j} in Equation 4 and recording the location (worker) of the one with the smallest neighbor set.
- `Propose`: Attaching the smallest neighbor set to p as $(p; C(v_{i+1}))$.
- `Intersect`: Sending $(p; C(v_{i+1}))$ to the worker that maintains each u_{k_j} and update $C(v_{i+1}) = C(v_{i+1}) \cap \mathcal{N}_G(u_{k_j})$.

After intersection, we will expand p by pushing into p every vertex of $C(v_{i+1})$.

Implementation Details. We directly use the authors' implementation [6], but slightly modify the codes to use the common graph data structure. We do not consider the dynamic version of BiGJoin in this paper, as the other strategies currently only support static context. The matching order is determined using a greedy heuristic that starts with the vertex of the largest degree, and consequently selects the next vertex with the most connections (id as tie breaker) with already-selected vertices.

[5] `https://github.com/TimelyDataflow/timely-dataflow/blob/master/examples/hashjoin.rs`

3.3 ShrCube

ShrCube strategy treats the join processing of the query Q as a hypercube of $n = |V_Q|$ dimension. It attempts to divide the hypercube evenly across the workers in the cluster, so that each worker can complete its own share without data communication. However, it is normally required that each data tuple is duplicated into multiple workers. This renders a space requirement of $\frac{M}{w^{1-\rho}}$ for each worker, where M is size of the input data, w is the number of workers and $0 < \rho \le 1$ is a query-dependent parameter. When ρ is close to 1, the algorithm ends up with maintaining the whole input data in each worker.

MultiwayJoin. MultiwayJoin applies the ShrCube strategy to solve subgraph matching in one single round. Consider w workers in the cluster, a query graph Q with $V_Q = \{v_1, v_2, \ldots, v_n\}$ vertices and $E_Q = \{e_1, e_2, \ldots, e_m\}$, where $e_i = (v_{i_1}, v_{i_2})$. Regarding each query vertex v_i, assign a positive integer as bucket number b_i that satisfies $\prod_{i=1}^{n} b_i = w$. The algorithm then divides the candidate data vertices for v_i evenly into b_i parts via a hash function $h : u \mapsto z_i$, where $u \in V_G, 1 \le z_i \le b_i$. This accordingly divides the whole computation into w shares, each of which can be indiced via an n-ary tuple $(z_1, z_2, \cdots, z_n)$, and is assigned to one worker. Afterwards, regarding each query edge $e_i = (v_{i_1}, v_{i_2})$, MultiwayJoin maps a data edge (u, u') as $(z_1, \cdots, z_{i_1} = h(u), \cdots, z_{i_2} = h(u'), \ldots, z_n)$, where other than z_{i_1} and z_{i_2}, each above z_i iterates through $\{1, 2, \cdots, b_i\}$, and the edge will be routed to the workers accordingly. Taking triangle query with $E_Q = \{(v_1, v_2), (v_1, v_3), (v_2, v_3)\}$ as an example. According to [13], $b_1 = b_2 = b_3 = b = \sqrt[n]{w}$ is an optimal bucket number assignment. Each edge (u, u') is then routed to the workers as: (1) $(h(u), h(u'), z)$ regarding (v_1, v_2); (2) $(h(u), z, h(u'))$ regarding (v_1, v_3); (3) $(z, h(u), h(u'))$ regarding (v_2, v_3), where the above z iterates through $\{1, 2, \cdots, b\}$. Consequently, each data edge is duplicated by roughly $3\sqrt[3]{w}$ times, and by expectation each worker will receive $\frac{3M}{w^{1-1/3}}$ edges. For unlabelled matching, MultiwayJoin utilizes the partial order of the query graph (Section 2.1) to reduce edge duplication, and details can be found in [13].

Implementation Details. There are two main impact factors of the performance of ShrCube. Firstly, the hypercube sharing by assigning proper b_i for v_i. Beame et al. [16] generalized the problem of computing optimal hypercube sharing for arbitrary query as linear programming. However, the optimal solution may assign fractional bucket number that is unwanted in practice. An easy refinement is to round down to an integer, but it will apparently result in idle workers. Chu et al. [21] addressed this issue via "Hypercube Optimization", that is to enumerate all possible bucket sequences around the optimal solutions, and choose the one that produces shares (product of bucket numbers) closest to the number of workers. We adopt this strategy in our implementation.

Secondly, the local algorithm. When the edges arrive at the worker, we collect them into a local graph (duplicate edges are removed), and use local algorithm to compute the matches. For unlabelled matching. we study the state-of-the-art local algorithms from "EmptyHeaded" [12] and "DualSim"[35]. "EmptyHeaded" is inspired by Ngo's worst-case optimal algorithm [43] that decomposes the query graph via "Hyper-Tree Decomposition", computes each decomposed part using worst-case optimal join and finally glues all parts together using hash join. "DualSim" was proposed by [35] for subgraph matching in the external-memory setting. The

idea is to first compute the matches of V_Q^{cc}, then the remaining vertices $V_Q \setminus V_Q^{cc}$ can be efficiently matched by enumerating the intersection of V_Q^{cc}'s neighbors. We find out that "DualSim" actually produces the same query plans as "EmptyHeaded" for all our benchmarking queries (Figure 4) except q_9. We implement both algorithms for q_9 and "DualSim" performs better than "EmptyHeaded" on the GO, US, GP and LJ datasets (Table 2). As a result, we adopt "DualSim" as the local algorithm for MultiwayJoin. For labelled matching, we implement "CFLMatch" proposed in [17] that has been shown so far to have the best performance.

Now we let each worker independently compute matches in its local graph. Simply doing so will result in duplicates, so we process deduplication as follows: given a match f that is computed in the worker identified by t_w, we can recover the tuple t_e^f of the matched edge $(f(v), f(v'))$ regarding the query edge $e = (v, v')$, then the match f is retained if and only if $t_w = t_e^f$ for every $e \in E_Q$. To explain this, let's consider $b = 2$, and a match $\{u_0, u_1, u_2\}$ for a triangle query (v_0, v_1, v_2), where $h(u_0) = h(u_1) = h(u_2) = 0$. It is easy to see that the match will be computed in workers of $(0, 0, 0)$ and $(0, 0, 1)$, while the match in worker $(0, 0, 1)$ will be eliminated as (u_0, u_2) that matches the query edge (v_0, v_2) can not be hashed to $(0, 0, 1)$ regarding (v_0, v_2). We can also avoid deduplication by separately maintaing each edge regarding different query edges it stands for, and use the local algorithm proposed in [21], but it results in too many edge duplicates that drain our memory even when processing a medium-size graph.

3.4 Others

PSgL and its implementation. PSgL iteratively processes subgraph matching via breadth-first traversal. All query vertices are configured three status, "white" (initialized), "gray" (candidate) and "black" (matched). Denote v_i as the vertex to match in the i^{th} round. The algorithm starts from matching initial query vertex v_1, and coloring the neighbors as "gray". In the i^{th} round, the algorithm applies the workload-aware expanding strategy at runtime, that is to select the v_i to expand among all current "gray" vertices based on a greedy heuristic to minimize the communication cost [48]; the partial results from previous round R_{i-1} (specially, $R_0 = \emptyset$) will be distributed among the workers based on the candidate data vertices that can match v_i; in the certain worker, the algorithm computes R_i by merging R_{i-1} with the matches of the `Star` formed by v_i and its "white" neighbors $\mathcal{N}_Q^w(v_i)$, namely $\texttt{Star}(v_i; \mathcal{N}_Q^w(v_i))$; after v_i is matched, v_i is colored as "black" and its "white" neighbors will be colored as "gray"; essentially, this process is analogous to StarJoin by processing $R_i = R_{i-1} \bowtie \texttt{Star}(v_i; \mathcal{N}_Q^w(v_i))$. Thus, PSgL can be seen as an alternative implementation of StarJoin on `Pregel` [40]. In this work, we also implement PSgL using a `Pregel` on `Timely`. Note that we introduce `Pregel` api to as much as possible replay the implementation of PSgL. In fact, it is simply wrapping `Timely`'s primitive operators such as `binary_notify` and `loop` [6], and barely introduces extra cost to the implementation. Our experimental results demonstrate similar findings as prior work [38] that PSgL's performance is dominated by CliqueJoin [38]. Thus, we will not further discuss this algorithm in this paper.

[6] `https://github.com/frankmcsherry/blog/blob/master/posts/2015-09-21.md`

CrystalJoin and its implementation. CrystalJoin aims at resolving the "output crisis" by compressing the results of subgraph matching [44]. The authors defined a structure called *crystal*, denoted $\mathcal{Q}(x, y)$. A crystal is a subgraph of Q that contains two sets of vertices V_x and V_y ($|V_x| = x$ and $|V_y| = y$), where the induced subgraph $Q(V_x)$ is a x-clique, and every vertex in V_y connects to all vertices of V_x. We call V_x clique vertices, and V_y the bud vertices. The algorithm first obtains the minimum vertex cover V_Q^c, and then applies the *Core-Crystal Decomposition* to decompose the query graph into the *core* $Q(V_Q^c)$ and a set of *crystal*s $\{\mathcal{Q}_1(x_1, y_1), \ldots, \mathcal{Q}_t(x_t, y_t)\}$. The crystals must satisfy that $\forall 1 \le i \le t$, $Q(V_{x_i}) \subseteq Q(V_Q^c)$, namely, the clique part of each crystal is a subgraph of the core. As an example, we plot a query graph and the corresponding core-crystal decomposition in Figure 3. Note that in the example, both crystals have an edge (i.e. 2-clique) as the clique part.

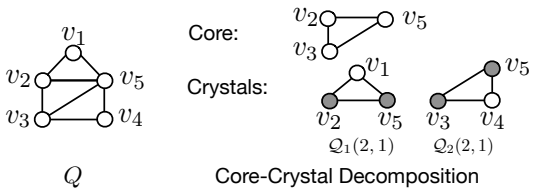


Figure 3: The Core-Crystal Decomposition of the query graph.

With core-crystal decomposition, the computation has accordingly split into three stages:

1. **Core computation**. Given that $Q(V_Q^c)$ itself is a query graph, the algorithm can be recursively applied to compute $Q(V_Q^c)$ according to [44].
2. **Crystal computation**. A special case of crystal is $\mathcal{Q}(x, 1)$, which is indeed a $(x + 1)$-clique. Suppose an instance of the $Q(V_x)$ is $f_x = \{u_1, u_2, \ldots, u_x\}$, we can represent the matches w.r.t. f_x as (f_x, I_y), where $I_y = \bigcap_{i=1}^{x} \mathcal{N}_G(u_i)$ denotes the set of vertices that can match V_y. This can naturally be extended to the case with $y > 1$, where any y-combinations of the vertices of I_y together with f_x represent a match. This way, the matches of crystals can be largely compressed.
3. **One-time assembly**. This stage assembles the core instances and the compressed crystal matches to produce the final results. More precisely, this stage is to **join** the core instance with the crystal matches.

We notice two technical obstacles to implement CrystalJoin according to the paper. Firstly, it is worth noting that the core $Q(V_Q^c)$ may be disconnected, a case that can produce exponential number of results. The authors applied a query-specific optimization in the original implementation to resolve this issue. Secondly, the authors proposed to precompute the cliques up to certain k, while it is often cost-prohibitive to do so in practice. Take UK (Table 2) dataset as an example, the triangles, 4-cliques and 5-cliques are respectively about 20, 600 and 40000 times larger than the graph itself. It is worth noting that the main purpose of this paper is not to study how well each algorithm performs for a specific query, which has its theoretical value, but can barely guide practice. After communicating with the authors, we adapt CrystalJoin in the following. Firstly, we replace the core $Q(V_Q^c)$ with the induced subgraph of the minimum connected vertex cover $Q(V_Q^{cc})$. Secondly, instead of implementing CrystalJoin as a strategy,

we use it as an alternative join plan (matching order) for WOPTJOIN. According to CrystalJoin, we first match V_Q^{cc}, while the matching order inside and outside V_Q^{cc} still follows WOPTJOIN's greedy heuristic (Section 3.2). It is worth noting that this adaptation achieves high performance comparable to the original implementation. In fact, we also apply CrystalJoin plan to BINJOIN, while it does not perform as well as the WOPTJOIN version, thus we do not discuss this implementation.

FullRep and its implementation. FULLREP simply maintains a full replica of the graph in each physical machine. Each worker picks one independent share of computation and solves it using existing local algorithm.

The implementation is straightforward. We let each worker pick its share of computation via a Round-Robin strategy, that is we settle an initial query vertex v_1, and let first worker match v_1 with u_1 to continue the remaining process, and second worker match v_1 with u_2, and so on. This simple strategy already works very well on balancing the load of our benchmarking queries (Figure 4). We use "DualSim" for unlabelled matching and "CFLMatch" for labelled matching as MultiwayJoin.

3.5 Worst-case Optimality.

Given a query Q and the data graph G, we denote the maximum possible result set as $\overline{R}_G(Q)$. Simply speaking, an algorithm is worst-case optimal if the aggregation of the total intermediate results is bounded by $\Theta(|\overline{R}_G(Q)|)$. Ngo et al. proposed a class of worst-case optimal join algorithm called `GenericJoin` [43], and we first overview this algorithm.

GenericJoin. Let the join be $R(V) = \bowtie_{F\subseteq\Psi} R(F)$, where $\Psi = \{U \mid U \subseteq V\}$ and $V = \bigcup_{U\in\Psi} U$. Given a vertex subset $U \subseteq V$, let $\Psi_U = \{V' \mid V' \in \Psi \wedge V' \cap U \neq \emptyset\}$, and for a tuple $t \in R(V)$, denote t_U as t's projection on U. We then show the `GenericJoin` in Algorithm 1.

Algorithm 1: `GenericJoin`$(V, \Psi, \bowtie_{U\in\Psi} R(U))$

```
1 R(V) ← ∅;
2 if |V| = 1 then
3     Return ⋃_{U∈Ψ} R(U);
4 V ← (I, J), where ∅ ≠ I ⊂ V, and J = V \ I;
5 R(I) ← GenericJoin(I, Ψ_I, ⋈_{U∈Ψ_I} π_I(R(U)));
6 forall t_I ∈ R(I) do
7     R(J)_{w.r.t. t_I} ← GenericJoin(J, Ψ_J, ⋈_{U∈Ψ_J}
        π_J(R(U) ⋉ t_I));
8     R(V) ← R(V) ∪ {t_I} × R(J)_{w.r.t. t_I};
9 Return R(V);
```

In Algorithm 1, the original join is recursively decomposed into two parts $R(I)$ and $R(J)$ regarding the disjoint sets I and J. From line 5, it is clear that $R(I)$ will record $R(V)$'s projection on I, thus we have $|R(I)| \leq |\overline{R}(V)|$, where $\overline{R}(V)$ is the maximum possible results of the query. Meanwhile, in line 7, the semi-join $R(U) \ltimes t_I = \{r \mid r \in R(U) \wedge r_{(U\cup I)} = t_{(U\cup I)}\}$ only retains those $R(J)$ w.r.t. t_I that can end up in the join result, which infers that the $R(J)$ must also be bounded by the final results. This intuitively explains the worst-case optimality of `GenericJoin`, while we refer interested readers to [43] for a complete proof.

It is easy to see that BiGJoin is worst-case optimal. In Algorithm 1, we select I in line 4 by popping the edge relation $E(v_s, v_i)(s < i)$ in the i^{th} step. In line 7, the recursive call to solve the semi-join $R(U) \ltimes t_I$ actually corresponds to the intersection process.

Worst-case Optimality of CliqueJoin. Note that the two clique relations in Equation 3 interleave one common edge (v_2, v_4) in the query graph. This optimization, called "overlapping decomposition" [38], eventually contributes to CliqueJoin's worst-cast optimality. Note that it is not possible to apply this optimization to StarJoin and TwinTwigJoin. We have the following theorem.

THEOREM 1. CliqueJoin *is worst-case optimal while applying "overlapped decomposition".*

PROOF. We implement CliqueJoin using Algorithm 1 in the following. Note that $Q(V)$ denotes a subgraph of Q induced by V. In line 2, we change the stopping condition to "$Q(I)$ is either a clique or a star". In line 4, the I is selected such that $Q(I)$ is either a clique or a star. Note that by applying the "overlapping decomposition" in CliqueJoin, the sub-query of the J part must be the J-induced graph $Q(J)$, and it will also include the edges of $E_{Q(I)} \cap E_{Q(J)}$, which infers that $R(Q(J)) = R(Q(J)) \ltimes R(Q(I))$, and just reflects the semi-join in line 7. Therefore, CliqueJoin belongs to `GenericJoin`, and is thus worst-case optimal. □

4. OPTIMIZATIONS

We introduce the three general-purpose optimizations, `Batching`, `TrIndexing` and `Compression` in this section, and how we orthogonally apply them to BINJOIN and WOPTJOIN algorithms. In the rest of the paper, we will use the strategy BINJOIN, WOPTJOIN, SHRCUBE instead of their corresponding algorithms, as we focus on strategy-level comparison.

4.1 Batching

Let $R(V_i)$ be the partial results that match the given vertices $V_i = \{v_{s_i}, v_{s_2}, \ldots, v_{s_i}\}$ (R_i for short if V_i follows a given order), and $R(V_j)$ denote the more complete results with $V_i \subset V_j$. Denote $R_j|R_i$ as the tuples in R_j whose projection on V_i equates R_i. Let's partition R_i into b **disjoint** parts $\{R_i^1, R_i^2, \ldots, R_i^b\}$. We define `Batching` on $R_j|R_i$ as the technique to independently process the following subtasks that compute $\{R_j|R_i^1, R_j|R_i^2, \ldots, R_j|R_i^b\}$. Obviously, $R_j|R_i = \bigcup_{k=1}^{b} R_j|R_i^k$.

WOptJoin. Recall from Section 3.2 that WOPTJOIN progresses according to a predefined matching order $\{v_1, v_2, \ldots, v_n\}$. In the i^{th} round, WOPTJOIN will `Propose` on each $p \in R_{i-1}$ to compute R_i. It is not hard to see that we can easily apply `Batching` to the computation of $R_i|R_{i-1}$ by randomly partitioning R_{i-1}. For simplicity, the authors implemented `Batching` on $R(Q)|R_1(v_1)$. Note that $R_1(v_1) = V_G$ in unlabelled matching, which means that we can achieve `Batching` simply by partitioning the data vertices[7]. For short, we also say the strategy batches on v_1, and call v_1 the batching vertex. We follow the same idea to apply `Batching` to BINJOIN algorithms.

BinJoin. While it is natural for WOPTJOIN to batch on v_1, it is non-trivial to pick such a vertex for BINJOIN. Given a decomposition of the query graph $\{p_1, p_2, \ldots, p_s\}$, where each p_i is a join unit, we have $R(Q) = R(p_1) \bowtie R(p_2) \cdots \bowtie R(p_s)$. If we partition $R_1(v)$ so as to batch on $v \in V_Q$, we

[7]Practically, it is more efficient to start from matching the edges instead of the vertices, and we can batch on $R(Q)|R_2$, where $R_2 = E_G$.

correspondingly split the join task, and one of the sub-task is $R(Q)|R_1^k(v) = R(p_1)|R_1^k(v) \bowtie \cdots \bowtie R(p_s)|R_1^k(v)$ ($R_1^k(v)$ is one partition of $R_1(v)$). Observe that if there exists a join unit p where $v \notin V_p$, we must have $R(p) = R(p)|R_1^k(v)$, which means $R(p)$ have to be fully computed in each sub-task. Let's consider the example query in Equation 2.

$$R(Q) = T_1(v_1, v_2, v_4) \bowtie T_2(v_2, v_3, v_4) \bowtie T_3(v_3, v_4).$$

Suppose we batch on v_1, the above join can be divided into the following independent sub-tasks:

$$R(Q)|R_1^1(v_1) = (T_1(v_1, v_2, v_4)|R_1^1(v_1)) \bowtie T_2(v_2, v_3, v_4) \bowtie T_3(v_3, v_4),$$
$$R(Q)|R_1^2(v_1) = (T_1(v_1, v_2, v_4)|R_1^2(v_1)) \bowtie T_2(v_2, v_3, v_4) \bowtie T_3(v_3, v_4),$$
$$\cdots$$
$$R(Q)|R_1^b(v_1) = (T_1(v_1, v_2, v_4)|R_1^b(v_1)) \bowtie T_2(v_2, v_3, v_4) \bowtie T_3(v_3, v_4).$$

It is not hard to see that we will have to re-compute $T_2(v_2, v_3, v_4)$ and $T_3(v_3, v_4)$ in all the above sub-tasks. Alternatively, if we batch on v_4, we can avoid such re-computation as T_1, T_2 and T_3 can all be partitioned in each sub-task. Inspired by this, for BinJoin, we come up with the heuristic to apply `Batching` on the vertex that presents in as many join units as possible. Note that such vertex can only be in the join key, as otherwise it must at least not present in one side of the join. For complex query, we can still have join unit that does not contain any vertex for `Batching` after applying the above heuristic. In this case, we either re-compute the join unit, or cache it on disk. Another problem caused by this is potential memory burden of the join. Thus, we devise the *join-level* `Batching` following the idea of external `MergeSort`. Specifically, we inject a `Buffer-and-Batch` operator for the two data streams before they arrive at the `Join` operator. `Buffer-and-Batch` functions in two parts:

- `Buffer`: While the operator receives data from the upstream, it buffers the data until reaching a given threshold. Then the buffer is sorted according to the join key's hash value and spilled to the disk. The buffer is reused for the next batch of data.
- `Batch`: After the data to join is fully received, we read back the data from the disk in a batching manner, where each batch must include all join keys whose hash values are within a certain range.

While one batch of data is delivered to the `Join` operator, `Timely` allows us to supervise the progress and hold the next batch until the current batch completes. This way, the internal memory requirement is one batch of the data. Note that such join-level `Batching` is natively implemented in Hadoop's "Shuffle" stage, and we replay this process in `Timely` to improve the scalability of the algorithm.

4.2 Triangle Indexing

As the name suggests, `TrIndexing` precomputes the triangles of the data graph and indices them along with the graph data to prune infeasible results. The authors of BiGJoin [14] optimized the 4-clique query by using the triangles as base relations to join, which reduces the rounds of join and network communication. In [44], the authors proposed to not only maintain triangles, but all k-cliques up to a given k. As we mentioned earlier, it incurs huge extra cost of maintaining triangles already, let alone larger cliques.

In addition to the default hash partition, Lai et al. proposed "triangle partition" [38] by also incorporating the edges among the neighbors (it forms triangles with the anchor vertex) in the partition. "Triangle partition" allows BinJoin to use clique as the join unit [38], which greatly reduces the intermediate results of certain queries and improves the performance. "Triangle partition" is in de facto a variant of `TrIndexing`, which instead of explicitly materializing the triangles, maintains them in the local graph structure (e.g. adjacency list). As we will show in the experiment (Section 5), this will save a lot of space compared to explicit triangle materialization. Therefore, we adopt the "triangle partition" for `TrIndexing` optimization in this work.

BinJoin. Obviously, BinJoin becomes CliqueJoin with `TrIndexing`, and StarJoin (or TwinTwigJoin) otherwise. With worst-case optimality guarantee (Section 3.5), BinJoin should perform much better with `TrIndexing`, which is also observed in "Exp-1" of Section 5.

WOptJoin. In order to match v_i in the i^{th} round, WOptJoin utilizes `Count`, `Propose` and `Intersect` to process the intersection of Equation 4. For ease of presentation, suppose v_{i+1} connects to the first s query vertices $\{v_1, v_2, \ldots, v_s\}$, and given a partial match, $\{f(v_1), \ldots, f(v_s)\}$, we have $C(v_{i+1}) = \bigcap_{j=1}^{s} \mathcal{N}_G(f(v_j))$. In the original implementation, it is required to send $(p; C(v_{i+1}))$ via network to all machines that contain each $f(v_j)(1 \le j \le s)$ to process the intersection, which can render massive communication cost. In order to reduce the communication cost, we implement `TrIndexing` for WOptJoin in the following. We first group $\{v_1, \ldots, v_s\}$ such that for each group $U(v_x)$, we have

$$U(v_x) = \{v_x\} \cup \{v_y \mid (v_x, v_y) \in E_Q\}.$$

Because of `TrIndexing`, we have $\mathcal{N}_G(f(v_y))$ $(\forall v_y \in U(v_x))$ maintain in $f(v_x)$'s partition. Thus, we only need to send the prefix to $f(v_x)$'s machine, and the intersection within $U(v_x)$ can be done locally. We process the grouping using a greedy strategy that always constructs the largest group from the remaining vertices.

Remark 1. The "triangle partition" may result in maintaining a large portion of the data graph in certain partition. Lai et al. pointed out this issue, and proposed a space-efficient alternative by leveraging the vertex orderings [38]. That is, given the partitioned vertex as u, and two neighbors u' and u'' that close a triangle, we place the edge (u', u'') in the partition only when $u < u' < u''$. Although this alteration reduces storage, it may affect the effectiveness of `TrIndexing` for WOptJoin and the implementations of `Batching` and `Compression` for BinJoin algorithms. Take WOptJoin as an example, after using the space-efficient "triangle partition", we should modify the above grouping as:

$$U(v_x) = \{v_x\} \cup \{v_y \mid (v_x, v_y) \in E_Q \wedge (v_x, v_y) \in O_Q\}.$$

Note that the order between query vertices are for symmetry breaking (Section 2.1), and it may not present in certain query, which makes `TrIndexing` completely useless for WOptJoin.

4.3 Compression

Subgraph matching is a typical combinatorial problem, and can easily produce results of exponential size. `Compression` aims to maintain the (intermediate) results in a compressed form to reduce resource allocation and communication cost. In the following, when we say "compress a query vertex", we mean maintaining its matched data vertices in the form of an array, instead of unfolding them in line

with the one-one mapping of a match (Definiton 1). Qiao et al. proposed CrystalJoin to study `Compression` in general for subgraph matching. As we introduced in Section 3.4, CrystalJoin first extracts the minimum vertex cover as uncompressed part, and then it can compress the remaining query vertices as the intersection of certain uncompressed matches' neighbors. Such `Compression` leverages the fact that all dependencies (edges) of the compressed part that requires further computation are already covered by the uncompressed part, thus it can stay compressed until the actual matches are requested. CrystalJoin inspires a heuristic for doing `Compression`, that is *to compress the vertices whose matches will* ***not*** *be used in any future computation.* In the following, we will apply the same heuristic to the other algorithms.

BinJoin. Obviously we can not compress any vertex that presents in the join key. What we need to do is to simply locate the vertices to compress in the join unit, namely star and clique. For star, the root vertex must remain uncompressed, as the leaves' computation depends on it. For clique, we can only compress one vertex, as otherwise the mutual connection between the compressed vertices will be lost. In a word, we compress two types of vertices for BinJoin, (1) non-key and non-root vertices of a star join unit, (2) one non-key vertex of a clique join unit.

WOptJoin. Based on a predefined join order $\{v_1, v_2, \ldots, v_n\}$, we can compress v_i $(1 \leq i \leq n)$, if there does not exist v_j $(i < j)$ such that $(v_i, v_j) \in E_Q$. In other words, v_i's matches will never be involved in any future intersection (computation). Note that v_n's can be trivially compressed. With `Compression`, when v_i is compressed, we will maintain its matches as an array instead of unfolding it into the prefix like a normal vertex.

5. EXPERIMENTS

5.1 Experimental settings

Environments. We deploy two clusters for the experiments: (1) a local cluster of 10 machines connected via one 10GBps switch and one 1GBps switch. Each machine has 64GB memory, 1 TB disk and 1 Intel Xeon CPU E3-1220 V6 3.00GHz with 4 physical cores; (2) an AWS cluster of 40 "r5-2xlarge" instances connected via a 10GBps switch, each with 64GB memory, 8 vCpus and 500GB Amazon EBS storage. By default we use the local cluster of 10 machines with 10GBps switch. We run 3 workers in each machine in the local cluster, and 6 workers in the AWS cluster for `Timely`. The codes are implemented based on the open-sourced `Timely` dataflow system [9] using Rust 1.32. We are still working towards open-sourcing the codes, and the bins together with their usages are temporarily provided[8] to verify the results.

Metrics. In the experiments, we measure query time T as the slowest worker's wall clock time from an average of three runs. We allow 3 hours as the maximum running time for each test. We use `OT` and `OOM` to indicate a test case runs out of the time limit and out of memory, respectively. By default we will not show the `OOM` results for clear presentation. We divide T into two parts, the computation time T_{comp} and the communication time T_{comm}. We measure T_{comp} as the time the slowest worker spends on actual computation by timing every computing function. We are aware that the actual communication time is hard to measure as `Timely` overlaps computation and communication to improve throughput. We consider $T - T_{comp}$, which mainly records the time the worker waits data from the network channel (a.k.a. communication time). While the other part of communication that overlaps computation is of less interest as it does not affect the query progress. As a result, we simply let $T_{comm} = T - T_{comp}$ in the experiments. We measure the maximum peak memory using Linux's "`time -v`" in each machine. We define the communication cost as the number of integers a worker receives during the process, and measure the maximum communication cost among the workers accordingly.

Dataset Formats. We preprocess each dataset as follows: we treat it as a simple undirected graph by removing self-loop and duplicate edges, and format it using "Compressed Sparse Row" (CSR) [3]. We relabel the vertex id according to the degree and break the ties arbitrarily.

Compared Strategies. In the experiments, we implement BinJoin and WOptJoin with all `Batching`, `TrIndexing` and `Compression` optimizations (Section 4). ShrCube is implemented with "Hypercube Optimization" [21], and "DualSim" (unlabelled) [35] and "CFLMatch" (labelled) [17] as local algorithms. FullRep is implemented with the same local algorithms as ShrCube.

Auxiliary Experiments. We have also conducted several auxiliary experiments to study the strategies of BinJoin, WOptJoin, ShrCube and FullRep: (1) "Scalability" experiment in unlabelled matching. The experiment shows that FullRep scales out the best, followed by WOptJoin, BinJoin and ShrCube; (2) "Vary Density" experiment in labelled matching. This experiment does not give interesting findings; (3) "Vary Labels" experiment in labelled matching. This experiment shows the transition from unlabelled to labelled matching for all strategies. We present these experiments in the appendix of the full paper [4].

5.2 Unlabelled Experiments

Datasets. The datasets used in this experiment are shown in Table 2. All datasets except SY are downloaded from public source, which are indicated by the letter in the bracket (S [10], W [11], D [1]). All statistics are measured as G is an undirected graph. Among the datasets, GO is a small dataset to study cases of extremely large (intermediate) result set; LJ, UK and FS are three popular datasets used in prior works, featuring statistics of real social network and web graph; GP is the google plus ego network, which is exceptionally dense; US and EU, on the other end, are sparse road networks. These datasets vary in number of vertices and edges, densities and maximum degree, as shown in Table 2. We synthesize the SY data according to [19] that generates data with real-graph characteristics. Note that the data occupies roughly 80GB space, and is larger than the configured memory of our machine. We synthesize the data because we do not find public accessible data of this size. Larger dataset like Clueweb [2] is available, but it is beyond the processing power of our current cluster.

Each data is hash partitioned ("hash") across the cluster. We also implement the "triangle partition" ("tri.") for `TrIndexing` optimization (Section 4.2). To do so, we use BiGJoin to compute the triangles and send the triangle edges to corresponding partition. We record the time T_* and av-

[8] `https://goo.gl/Xp5BrW`

erage number of edges $|\overline{E_*}|$ of the two partition strategies. The partition statistics are recorded using the local cluster, except for SY that is processed in the AWS cluster. From Table 2, we can see that $|\overline{E_{tri.}}|$ is noticeably larger, around 1-10 times larger than $|\overline{E_{\text{hash}}}|$. Note that in GP and UK, which either is dense, or must contain a large dense community, the "triangle partition" can maintain a large portion of data in each partition. While compared to complete triangle materialization, "triangle partition" turns out to be much cheaper. For example, the UK dataset contains around 27B triangles, which means each partition in our local cluster should by average take 0.9B triangles (three integers); in comparison, UK's "triangle partition" only maintains an average of 0.16B edges (two integers) according to Table 2.

We use US, GO and LJ as default datasets in the experiments "Exp-1", "Exp-2" and "Exp-3" in order to collect useful feedbacks from successful queries, while we may not present certain cases when they do not give new findings.

Queries. The queries are presented in Figure 4. We also give the partial order under each query for symmetry breaking. The queries except q_7 and q_8 are selected based on all prior works [14, 36, 38, 44, 49], while varying in number of vertices, densities, and the vertex cover ratio $|V_Q^{cc}|/|V_Q|$, in order to better evaluate the strategies from different perspectives. The three queries q_7, q_8 and q_9 are relatively challenging given their result scale. For example, the smallest dataset GO contains $2,168$B(illion) q_7, 330B q_8 and $1,883$B q_9, respectively. For short of space, we record the number of results of each successful query on each dataset in the appendix. Note that q_7 and q_8 are absent from existing works, while we benchmark q_7 considering the importance of path query in practice, and q_8 considering the varieties of the join plans.

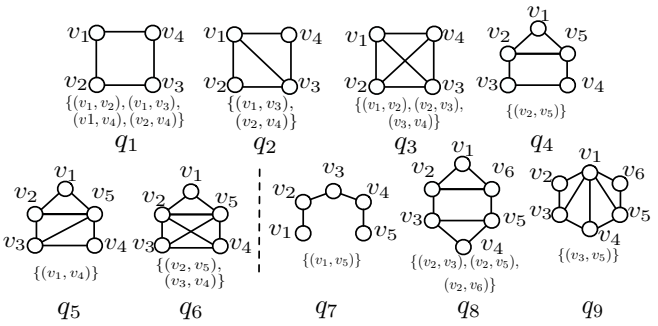


Figure 4: The unlabelled queries.

Exp-1: Optimizations. We study the effectiveness of `Batching`, `TrIndexing` and `Compression` for both BinJoin and WOptJoin strategies, by comparing BinJoin and WOptJoin with their respective variants with one optimization off, namely "NoBatching", "NoTrindexing" and "NoCompression". In the following, we use the suffix of "(w.o.b.)", "(w.o.t.)" and "(w.o.c.)" to represent the three variants. We use the queries q_2 and q_5, and the results of US and LJ are shown in Figure 5.

While comparing BinJoin with BinJoin(w.o.b.), we observe that `Batching` barely affects the performance of q_2, but severely for q_5 on LJ (1800s vs 4000s (w.o.b.)). The reason is that we still apply join-level `Batching` for BinJoin(w.o.b.) that dumps the intermediate data to the disk (Section 4.1). While q_5's intermediate data includes the massive results of sub-query $Q(\{v_2, v_3, v_4, v_5\})$, which incurs huge amount of disk I/O (US does not have this problem as it produces very few results). We also run q_5 without the join-level `Batching` on LJ, but it fails with `OOM`. For BinJoin, `TrIndexing` is a critical optimization, with the observed performance of BinJoin better than that of BinJoin(w.o.t.), especially so on LJ. This is expected as BinJoin(w.o.t.) actually degenerates to StarJoin. `Compression`, on the one hand, allows BinJoin to run much faster than BinJoin(w.o.c.) for both queries on LJ, on the other hand, makes it slower on US. The reason is that US is a sparse dataset with few room for `Compression`, while `Compression` itself incurs extra cost. We also compare BinJoin with BinJoin(w.o.c.) on the other sparse graph EU, and the results are the same.

For WOptJoin strategy, `Batching` has little impact to the performance. Surprisingly, after using `TrIndexing` to WOptJoin, the improvement by average is only around 18%. We do another experiment in the same cluster but using 1GBps switch, which shows WOptJoin is over 6 times faster than WOptJoin(w.o.t.) for both queries on LJ. Note that `Timely` uses separate threads to buffer received data from the network. Given the same computing speed, a faster network allows the data to be more fully buffered and hence less wait for the following computation. Similar to BinJoin, `Compression` greatly improve the performance while querying on LJ, but the opposite on US.

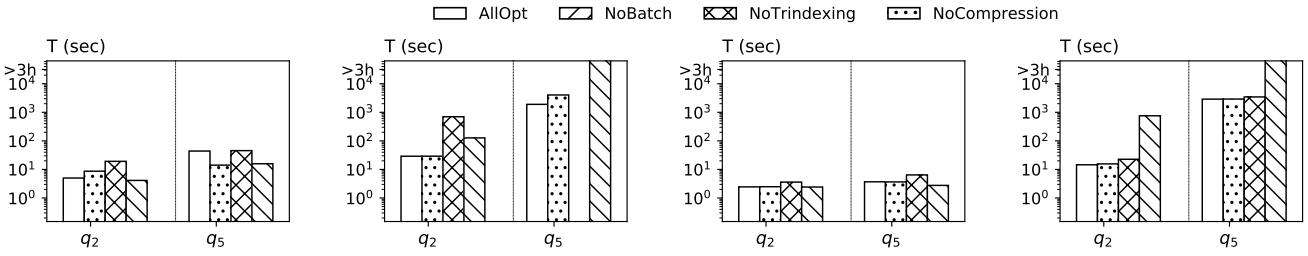
Exp-2 Join Plans. BinJoin and WOptJoin algorithms are sensitive to join plans. To improve the performance, BinJoin computes the join plan by minimizing the estimated cost of intermediate results, while WOptJoin follows the worst-case optimal join algorithm [43]. In this experiment, we will compare the optimal join plan P_0 adopted originally by the respective strategies, with an intuitively good plan P_1 selected based on our experience and a randomly generated plan P_2. Note that WOptJoin's P_1 is in de facto the CrystalJoin plan. The detailed join plans used in this experiment are given in the appendix. We use q_8 and q_9 due to the varieties of their join plans. We show the results of GO (US does not show much variance in results, while LJ fails most cases) in Table 3.

Observe that BinJoin's optimal join plan performs worse than P_1 for q_8. We find out that P_0 computes q_8 by joining two tailed triangles (triangle tailed with an edge), while P_1 joins the uppers "house-shape" sub-query with the bottom triangle. In theory, the tailed triangle has worse-case bound (AGM bound [43]) of $O(M^2)$, smaller than the house's $O(M^{2.5})$, and BinJoin's actually favors this plan based on cost estimation. However, we find out that the actual results of tailed triangle are very close to the houses on GO, which renders costly process for P_0 to join two tailed triangles. For q_9, P_0 performs over two times faster than P_1. Here, P_0 is scheduled to respectively compute the sub-queries $Q(\{v_1, v_2, v_3, v_4\})$ and $Q(\{v_1, v_4, v_5, v_6\})$, and join them in the last round. Alternatively, after computing the sub-query $Q(\{v_1, v_2, v_3, v_4\})$, P_1 grows the results by joining another triangle $\{v_1, v_4, v_5\}$, which produces the more costly sub-query $Q(\{v_1, v_2, v_3, v_4, v_5\})$.

As for WOptJoin, P_0 runs faster than P_1 while querying q_8, but slower while querying q_9. On the one hand, for q_8, the main difference between P_0 and P_1 is that: P_0 tends to close the upper triangle $\{v_1, v_2, v_6\}$, while P_1 needs to first compute the 2-path $\{v_2, v_3, v_5\}$ before closing the vertex-cover-induced sub-query $Q(V_Q^{cc})$. Typically, the triangle has lower cost than that of a 2-path. On the other hand, for q_9, after computing $Q(V_Q^{cc})$, namely the 2-path $\{v_1, v_3, v_5\}$, P_1 is able to compress all remaining vertices v_2, v_4 and v_6. In

Datasets	Name	$\|V_G\|$/mil	$\|E_G\|$/mil	$\overline{d}_G$	D_G	T_{hash}/s	$\|\overline{E_{\text{hash}}}\|$/mil	$T_{\text{tri.}}$/s	$\|\overline{E_{\text{tri.}}}\|$ /mil
google(S)	GO	0.86	4.32	5.02	6,332	1.53	0.28	2.31	1.23
gplus(S)	GP	0.11	12.23	218.2	20,127	5.57	0.80	46.5	10.68
usa-road(D)	US	23.95	28.85	2.41	9	12.43	1.89	3.69	1.90
livejournal(S)	LJ	4.85	43.37	17.88	20,333	14.25	2.81	20.33	12.49
uk2002(W)	UK	18.50	298.11	32.23	194,955	61.99	17.16	266.60	156.05
eu-road(D)	EU	173.80	342.70	3.94	20	72.96	22.47	16.98	22.98
friendster(S)	FS	65.61	1806.07	55.05	5,214	378.26	118.40	368.95	395.31
Synthetic	SY	372.00	10,000.00	53	613,461	2027	493.75	5604.00	660.61

Table 2: The unlabelled datasets.

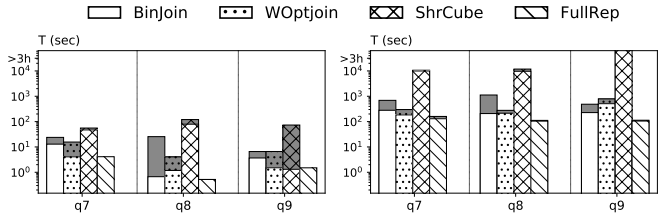


(a) BinJoin on US (b) BinJoin on LJ (c) WOptJoin on US (d) WOptJoin on LJ

Figure 5: Effectiveness of optimizations.

Test cases	P_0/s	P_1/s	P_2/s
BinJoin, q_8	1122	334	5175
BinJoin, q_9	485	1219	OT
WOptJoin, q_8	278	345	6180
WOptJoin, q_9	1039	788	OT

Table 3: Impact of different join plans (q_8 and q_9 on GO).



(a) US (b) GO

Figure 6: Challenging queries.

comparison, we can only compress v_2 and v_6 while using P_0. In this case, P_1 outperforms P_0 because it configures larger compression. In [44], the authors proved that it renders maximum compression to use the vertex cover as the uncompressed core. However, this may not necessarily result in the best performance, considering that it can be costly to compute the core part. In our experiments, the unlabelled q_4, q_8 and labelled q_8 are cases that CrystalJoin plan performs worse than the original BiGJoin plan (with `Compression` optimization), where CrystalJoin plan does not render larger compression while having to process costly core part. As a result, we only recommend CrystalJoin plan when it leads to strictly larger compression.

The random plan P_2 used in both BinJoin and WOptJoin are noticeably worse, because they all produce massive intermediate results.

It is very challenging to determine the optimal join plans for both BinJoin and WOptJoin because of many impact factors, which will be further discussed in Section 6. As there lacks in a systematic view, we still use their original "optimal" plan by default in the following, unless otherwise specified.

Exp-3 Challenging Queries. We study the challenging queries q_7, q_8 and q_9 in this experiment. We run this experiment using BinJoin, WOptJoin, ShrCube and FullRep, and show the results of US and GO (LJ failed all cases) in Figure 6. Recall that we split the time into computation time and communication time (Section 5.1), here we plot the communication time as gray filling in each bar of Figure 6.

FullRep beats all the other strategies, while ShrCube fails q_8 and q_9 on GO because of OT. Although ShrCube uses the same local algorithm as FullRep, it spends a lot of time on deduplication (Section 3.3).

We focus on comparing BinJoin and WOptJoin on GO dataset. On the one hand, WOptJoin outperforms BinJoin for q_7 and q_8. Their join plans of q_7 are nearly the same except that BinJoin relies on a global shuffling on v_3 to processing join, while WOptJoin sends the partial results to the machine that maintains the vertex to grow. It is hence reasonable to observe BinJoin's poorer performance for q_7 as shuffling is typically a more costly operation. The case of q_8 is similar and is not further discussed. On the other hand, even living with costly shuffling, BinJoin still performs better for q_9. WOptJoin's "optimal plan" actually follows BinJoin's P_1 in "Exp-2", which can not avoid the costly sub-query $Q(\{v_1, v_2, v_3, v_4, v_5\})$. On US dataset, WOptJoin consistently outperforms BinJoin for these queries. This is because that US does not produce massive intermediate results as LJ, thus BinJoin's shuffling cost consistently dominates.

Another observation is that the computation time dominates most of the evaluated cases, except BinJoin's q_8, WOptJoin and ShrCube's q_9 on US. We will further discuss this in Exp-4.

Exp-4 All-Around Comparisons. In this experiment, we run $q_1 - q_6$ using BinJoin, WOptJoin, ShrCube and FullRep across the datasets GP, LJ, UK, EU and FS. We also run WOptJoin with CrystalJoin plan in q_4 as it is the only query that renders different CrystalJoin plan from BiGJoin plan, and the results show that the performance with BiGJoin plan is consistently better. We report the re-

sults in Figure 7, where the communication time is plotted as gray filling. As a whole, among all 35 test cases, FullRep achieves the best 85% completion rate, followed by WOptJoin and BinJoin which complete 71.4% and 68.6% respectively, and ShrCube performs the worst with just 8.6% completion rate.

FullRep typically outperforms the other strategies. Observe that WOptJoin's performance is often very close to FullRep. The reason is that the WOptJoin's computing plans for these evaluated queries are similar to "DualSim" adopted by FullRep. The extra communication cost of WOptJoin has been reduced to very low while adopting `TrIndexing` optimization. While comparing WOptJoin with BinJoin, BinJoin is better for q_3, a clique query (join unit) that requires no join (a case of embarrassingly parallel). BinJoin performs worse than WOptJoin in most other queries, which, as we mentioned before, is due to the costly shuffling. There is an exception - querying q_1 on GP - where BinJoin performs better than both FullRep and WOptJoin. We explain this using our best speculation. GP is a very dense graph, where we observe nearly 100 vertices with degree around 10,000. To process q_1, after computing the sub-query $Q(\{v_1, v_2, v_4\})$, WOptJoin (and "DualSim") processes the intersection of v_1 and v_4 (their matches) for v_3. Those larger-degree vertices are now frequently pairing, leading to expensive intersection. In comparison, BinJoin computes q_1 by joining the sub-query $Q(\{v_1, v_2, v_3\})$ with $Q(\{v_1, v_3, v_4\})$. Because both strategies compute $Q(\{v_1, v_2, v_3\})$, we consider how BinJoin computes $Q(\{v_1, v_3, v_4\})$. BinJoin first locate the matched vertex of v_3, then matches v_1 and v_4 among its neighbors, which is generally cheaper than intersecting the neighbors of v_1 and v_4 to compute v_3. Due to the existence of these high-degree pairs, the cost WOptJoin's intersection can exceed BinJoin's shuffling.

We observe that the computation time T_{comp} dominates in most cases as we mentioned in "Exp-3". This is trivially true for ShrCube and FullRep, but it may not be clearly so for WOptJoin and BinJoin given that they all need to transfer a massive amount of intermediate data. We investigate this and find out two potential reasons. The first one attributes to `Timely`'s highly optimized communication component, which allows the computation to overlap communication by using extra threads to receive and buffer the data from the network so that it can be mostly ready for the following computation. The second one is the fast network. We re-run these queries using the 1GBps switch, while the results show the opposite trend that the communication time T_{comm} in turn takes over.

Exp-5 Web-Scale. We run the SY datasets in the AWS cluster of 40 instances. Note that FullRep can not be used as SY is larger than the machine's memory. We use the queries q_2 and q_3, and present the results of BinJoin and WOptJoin (ShrCube fails all cases due to `OOM`) in Table 4. The results are consistent with the prior experiments, but observe that the gap between BinJoin and WOptJoin while querying q_1 is larger. This is because that we now deploy 40 AWS instances, and BinJoin's shuffling cost increases.

5.3 Labelled Experiments

We use the LDBC social network benchmarking (SNB) [7] for labelled matching experiment due to the lack of labelled big graphs in the public. SNB provides a data generator that generates a synthetic social network of required statistics,

Queries	BinJoin (T_{comp})/s	WOptJoin (T_{comp})/s
q_2	8810 (6893)	1751 (1511)
q_3	76 (75)	518 (443)

Table 4: The web-scale queries.

Name	$\|V_G\|$	$\|E_G\|$	$\overline{d}_G$	D_G	# Labels
DG10	29.99	176.48	11.77	4,282,812	10
DG60	187.11	1246.66	13.32	26,639,563	10

Table 5: The labelled datasets.

and a document [8] that describes the benchmarking tasks, in which the complex tasks are actually subgraph matching. The join plans of BinJoin and WOptJoin for labelled experiments are generated as unlabelled case, but we use the label frequencies to break tie.

Datasets. We list the datasets and their statistics in Table 5. These datasets are generated using the "Facebook" mode with a duration of 3 years. The dataset's name, denoted as DGx, represents a scale factor of x. The labels are preprocessed into integers.

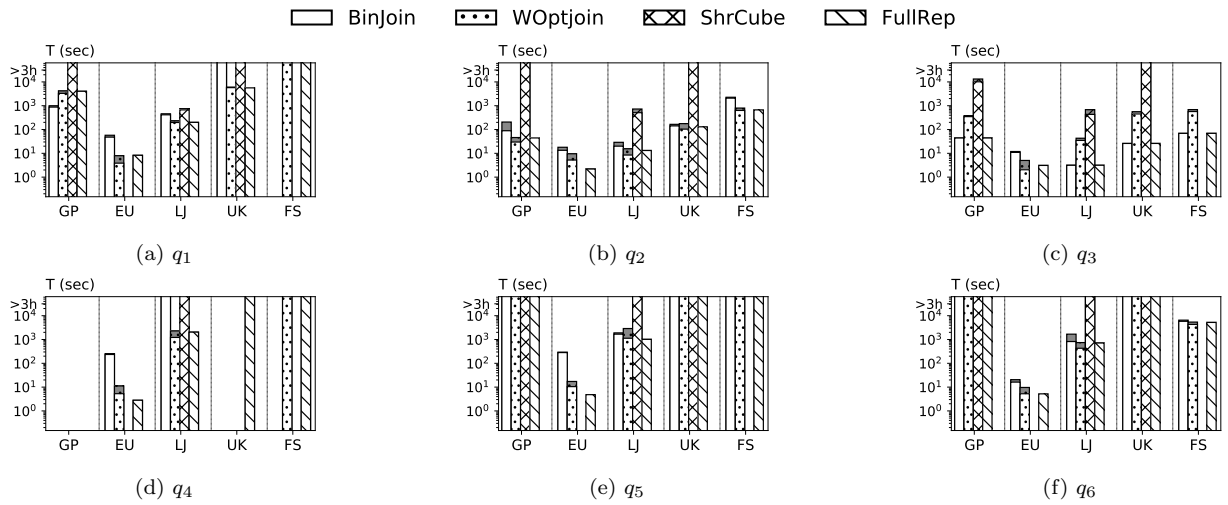
Queries. The queries, shown in Figure 8, are selected from the SNB's complex tasks with some adaptions, and the details are in the full paper.

Exp-6 All-Around Comparisons. We now conduct the experiment using all queries on DG10 and DG60, and present the results in Figure 9. Here we compute the join plans for BinJoin and WOptJoin by using the unlabelled method, but further using the label frequencies to break tie. The gray filling again represents communication time. FullRep outperforms the other strategies in many cases, except that it performs slightly slower than BinJoin for q_3 and q_5. This is because that q_3 and q_5 are join units, and BinJoin processes them locally in each machine as FullRep, and it does not build indices as "CFLMatch" used in FullRep. When comparing to WOptJoin, Among all these queries, we only have q_8 that configures different CrystalJoin plan (w.r.t. BiGJoin plan) for WOptJoin. The results show that the performance of WOptJoin drops about 10 times while using CrystalJoin plan. Note that the core part of q_8 is a 5-path of "Psn-City-Cty-City-Psn" with enormous intermediate results. As we mentioned in unlabelled experiments, it may not always be wise to first compute the vertex-cover-induced core.

We now focus on comparing BinJoin and WOptJoin. There are three cases that intrigue us. Firstly, observe that BinJoin performs much better than WOptJoin while querying q_4. The reason is high intersection cost as we discovered on GP dataset in unlabelled matching. Secondly, BinJoin performs worse than WOptJoin in q_7, which again is because of BinJoin's costly shuffling. The third case is q_9, the most complex query in the experiment. BinJoin performs much better while querying q_9. The bad performance of WOptJoin comes from the long execution plan together with costly intermediate results. The two algorithms all expand the three "Psn"s, and then grow via one of the "City"s to "Cty", but BinJoin approaches this using one join (a triangle $\bowtie$ a `TwinTwig`), while WOptJoin will first expand to "City" then further "Cty", and the "City" expansion is the culprit of the slower run.

6. DISCUSSIONS AND FUTURE WORK.

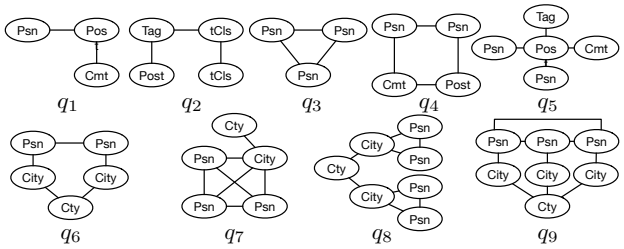
We discuss our findings and potential future work based on the experiments in Section 5. Eventually, we summarize



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

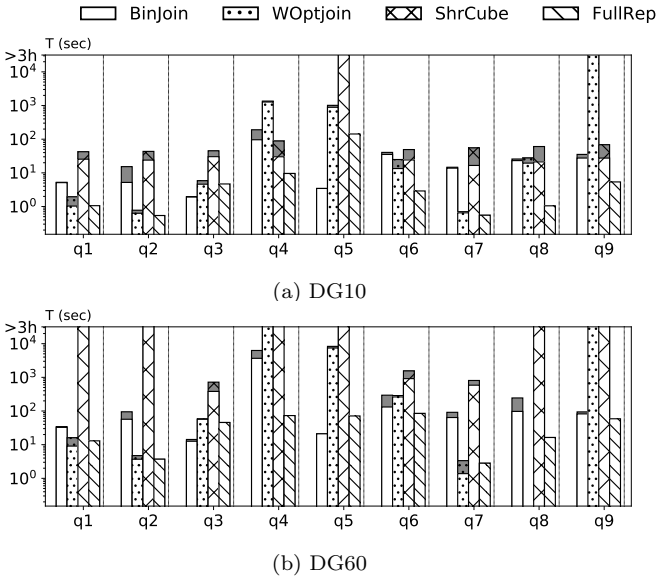
(d) q_4 (e) q_5 (f) q_6

Figure 7: All-around comparisons.



q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8 q_9

Figure 8: Labelled queries.



(a) DG10

(b) DG60

Figure 9: All-around comparisons of labelled matching.

the findings into a practical guide.

Strategy Selection. FullRep *is obviously the preferred choice when the machine can hold the graph data, while both* WOptJoin *and* BinJoin *are good alternatives when the graph is larger than the capacity of the machine.* For BinJoin and WOptJoin, on one side, BinJoin may perform worse than WOptJoin (e.g. unlabelled q_2, q_4, q_5) due to the expensive shuffling operation, on the other side, BinJoin can also outperform WOptJoin (e.g. unlabelled and labelled q_9) while avoiding costly sub-queries due to query decomposition. One way to choose between BinJoin and WOptJoin is to compare the cost of their respective join plans, and select the one with less cost. For now, we can either use cost estimation proposed in [38], or summing the worst-case bound, but none of them consistently gives the best solution, as will be discussed in "Optimal Join Plan". Alternatively, we refer to "EmptyHeaded" [12] to study a potential hybrid strategy of BinJoin and WOptJoin. Note that "EmptyHeaded" is developed in single-machine setting, and it does not take into consideration the impact of `Compression`, we hence leave such hybrid strategy in the distributed context as an interesting future work.

Optimizations. *Our experimental results suggest always using* `Batching`*, using* `TrIndexing` *when each machine has* ***sufficient*** *memory to hold "triangle partition", and using* `Compression` *when the data graph is not very sparse (e.g.* $\bar{d}_G \geq 5$*).* `Batching` often does not impact performance, so we recommend always using `Batching` due to the unpredictability of the size of (intermediate) results. `TrIndexing` is critical for BinJoin, and it can greatly improve WOptJoin by reducing communication cost, while it requires extra storage to maintain "triangle partition". Amongst the evaluated datasets, each "triangle partition" maintains an average of 30% data in our 10-machine cluster. Thus, we suggest a memory threshold of $60\%|E_G|$ (half for graph and half for running algorithm) for `TrIndexing` in a cluster of the same or larger scale. Note that the threshold does not apply to extremely dense graph. Among the three optimizations, `Compression` is the primary performance booster that improves the performance of BinJoin and WOptJoin by 5 times on average in all but the cases on the very sparse road networks. For such very sparse data graphs, `Compression` can render more cost than benefits.

Optimal Join Plan. *It is challenging to systematically determine the optimal join plans for both* BinJoin *and* WOptJoin. From the experiments, we identify three impact factors: (1) the worst-case bound; (2) cost estimation based on data statistics; (3) favoring the optimizations, especially `Compression`. All existing works only partially consider these factors, and we have observed sub-optimal join plans in the experiments. For example, BinJoin bases

the "optimal" join plan on minimizing the cost estimation, but the join plan does not render the best performance for unlabelled q_8; WOptJoin follows the worst-case optimality, while it may encounter costly sub-queries for labelled and unlabelled q_9; CrystalJoin focuses on maximizing the compression, while ignoring the facts that the vertex-cover-induced core part itself can be costly to compute. Additionally, there are other impact factors such as the partial orders of query vertices and the label frequencies, which have not been studied in this work due to short of space. It is another very interesting future work to thoroughly study the optimal join plan while considering all above impact factors.

Computation vs. Communication. *We argue that distributed subgraph matching nowadays is a computation-intensive task.* This claim holds when the cluster configures high-speed network (e.g. $\geq$ 10GBps), and the data processor can efficiently overlap computation with communication. Note that computation cost (either BinJoin's join or WOptJoin's intersection) is lower-bounded by the output size that is equal to the communication cost. Therefore, computation becomes the bottleneck if the network condition is good to guarantee the data to be delivered in time. Nowadays, the bandwidth of local cluster commonly exceeds 10GBps, and the overlapping of computation and communication is widely used in distributed systems (e.g. Spark [53], Flink [18]). As a result, we tend to see distributed subgraph matching as a computation-intensive task, and we advocate future research to devote more efforts into optimizing the computation while considering the following perspectives: (1) the new advancements of hardware, for example the co-processing on GPU in the coupled CPU-GPU architectures [29] and the SIMD programming model on modern CPU [31]; (2) general computing optimizations such as load balancing strategy and cache-aware graph data accessing [52].

A Practical Guide. Based on the experimental findings, we propose a practical guide for distributed subgraph matching in Figure 10. Note that this program guide is based on current progress of the literature, and future work is needed, for examples to study the hybrid strategy and the impact factors of the optimal join plan, before we can arrive at a solid decision-making to choose between BinJoin and WOptJoin.

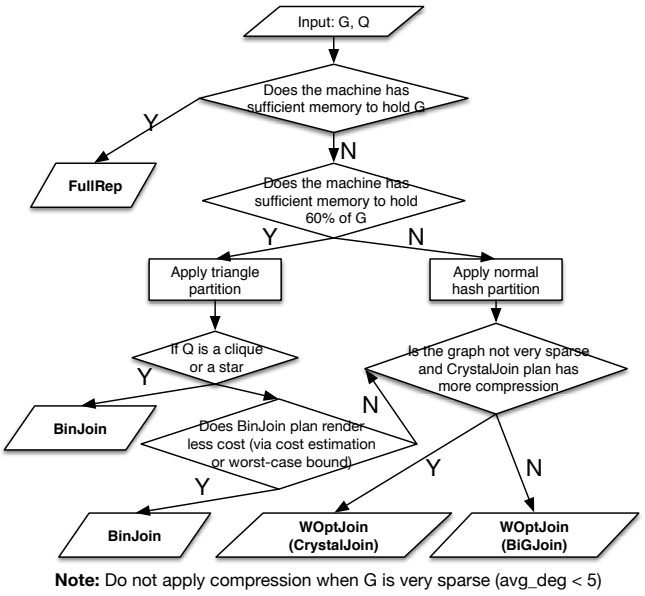


Figure 10: A practical guide of distributed subgraph matching.

7. RELATED WORK

Isomorphism-based Subgraph Matching. In the labelled case, Shang et al. [47] used the spanning tree of the query graph to filter infeasible results. Han et al. [28] observed the importance of matching order. In [45], the authors proposed to utilize the symmetry properties in the data graph to compress the results. Bi et al. [17] proposed CFLMatch based on the "core-forest-leaves" matching order, and obtained performance gain by postponing the notorious cartesian product.

The unlabelled case is also known as subgraph listing/enumeration, and due to the gigantic (intermediate) results, people have been either seeking scalable algorithms in parallel, or devising techniques to compress the results. Other than the algorithms studied in this paper (Section 3), Kim et al. proposed the external-memory-based parallel algorithm DualSim [35], which maintains the data graph in blocks on the disk, and matches the query graph by swapping in/out blocks of data to improve I/O efficiency.

Incremental Subgraph Matching. Computing subgraph matching in a continuous context has recently drawn a lot of attentions. Fan et al. [25] proposed incremental algorithm that identifies a portion of the data graph affected by the update regarding the query. The authors in [20] used the join scheme as BinJoin algorithms (Section 3.1). The algorithm maintained a left-deep join tree for the query, with each vertex maintaining a partial query and the corresponding partial results. Then one can compute the incremental answers of each partial query in response to the update, and utilizes the join tree to re-construct the results. Graphflow [34] solved incremental subgraph matching using join, in the sense that the incremental query can be transformed into m independent joins, where m is the number of query edges. Then they used the worst-case-optimal join algorithm to solve these joins in parallel. Most recently, Kim et al. proposed TurboFlux that maintains data-centric index for incremental queries, which achieves good tradeoff between performance and storage.

Query Languages and Systems. As the increasing demand of subgraph matching in graph analysis, people start to investigate easy-use and highly expressive subgraph matching language. Neo4j introduced *Cypher* [26], and now people are working on standardizing the semantics of subgraph matching based on Cypher [15]. Gradoop [33] is a system based on Apache Hadoop that translates a Cypher query into a MapReduce job. Aberger et al. proposed EmptyHeaded based on relational semantics for graph processing, in which they leveraged worst-case optimal join algorithm to solve subgraph matching. Arabesque [51] was designed to solve graph mining (continuously computing frequent subgraphs) at scale, while it can be configured for single subgraph query.

8. CONCLUSIONS

In this paper, we implement four strategies and three general-purpose optimizations for distributed subgraph matching based on `Timely` dataflow system, aiming for a

systematic, strategy-level comparisons among the state-of-the-art algorithms. Based on thorough empirical analysis, we summarize a practical guide, and we also motivate interesting future work for distributed subgraph matching.

9. REFERENCES

[1] The challenge9 datasets. http://www.dis.uniroma1.it/challenge9.

[2] The clubweb12 dataset. https://lemurproject.org/clueweb12.

[3] Compressed sparse row. https://en.wikipedia.org/wiki/Sparse_matrix.

[4] Distributed subgraph matching on timely dataflow - the full paper. https://goo.gl/zkTkL4.

[5] Giraph. http://giraph.apache.org/.

[6] The implementation of bigjoin. https://github.com/frankmcsherry/dataflow-join/.

[7] Ldbc benchmarks. http://ldbcouncil.org/benchmarks.

[8] The ldbc social network benchmark. https://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf.

[9] The open-sourced timely dataflow system. https://github.com/frankmcsherry/timely-dataflow.

[10] The snap datasets. http://snap.stanford.edu/data/index.html.

[11] The webgraph datasets. http://law.di.unimi.it/datasets.php.

[12] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. SIGMOD '16, pages 431–446.

[13] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *Proc. of ICDE'13*, 2013.

[14] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.*, 11(6):691–704, 2018.

[15] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5), 2017.

[16] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017.

[17] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. SIGMOD '16, pages 1199–1214, 2016.

[18] P. Carbone, A. Katsifodimos, . Kth, S. Sweden, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. 38, 01 2015.

[19] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

[20] S. Choudhury, L. B. Holder, G. Chin, K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *EDBT*, 2015.

[21] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. SIGMOD '15, pages 63–78.

[22] F. R. K. Chung, L. Lu, and V. H. Vu. The spectra of random graphs with given expected degrees. *Internet Mathematics*, 1(3), 2003.

[23] D. J. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? *SIGMOD Rec.*, 19(4):104–112.

[24] P. Erdos and A. Renyi. On the evolution of random graphs. In *Publ. Math. Inst. Hungary. Acad. Sci.*, 1960.

[25] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. SIGMOD '11, pages 925–936, 2011.

[26] S. Gorka and R. Philip. Improving first-party bank fraud detection with graph databases, 2016.

[27] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Proc. of RECOMB'07*, 2007.

[28] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. of SIGMOD'13*, 2013.

[29] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled cpu-gpu architectures. *Proc. VLDB Endow.*, 8(4):329–340, 2014.

[30] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. 2008.

[31] H. Inoue, M. Ohara, and K. Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proc. VLDB Endow.*, 8(3):293–304, 2014.

[32] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *SIGMOD'91*, pages 168–177, 1991.

[33] M. Junghanns, M. Kiesling, A. Averbuch, A. Petermann, and E. Rahm. Cypher-based graph pattern matching in gradoop. GRADES'17, pages 3:1–3:8.

[34] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An active graph database. SIGMOD '17, pages 1695–1698.

[35] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah. Dualsim: Parallel subgraph enumeration in a massive graph on a single machine. SIGMOD '16, pages 1231–1245, 2016.

[36] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10), 2015.

[37] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce: A cost-oriented approach. *The VLDB Journal*, 26(3):421–446, June 2017.

[38] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. *Proc. VLDB Endow.*, 10(3):217–228, 2016.

[39] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[40] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. of SIGMOD'10*, 2010.

[41] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? HOTOS'15, 2015.

[42] D. G. Murray, F. McSherry, R. Isaacs, M. Isard,

P. Barham, and M. Abadi. Naiad: A timely dataflow system. SOSP '13, pages 439–455, 2013.
[43] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3), 2018.
[44] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: On compression and computation. *Proc. VLDB Endow.*, 11(2):176–188, 2017.
[45] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proc. VLDB Endow.*, 8(5):617–628, 2015.
[46] R. Shamir and D. Tsur. Faster subtree isomorphism. In *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*, pages 126–131, 1997.
[47] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, 2008.
[48] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. SIGMOD '13, pages 505–516, 2013.
[49] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD'14*, pages 625–636. ACM, 2014.
[50] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9), 2012.
[51] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A system for distributed graph mining. SOSP '15, pages 425–440.
[52] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. SIGMOD '16, pages 1813–1828.
[53] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud'10*, pages 10–10.

APPENDIX

A. AUXILIARY EXPERIMENTS

Exp-7 Scalability of Unlabelled Matching. We vary the number of machines as 1, 2, 4, 6, 8, 10, and run the unlabelled queries q_1 and q_2 to see how each strategy (BinJoin, WOptJoin, ShrCube and FullRep) scales out. We further evaluate "Single Thread", a serial algorithm that is specially implemented for these two queries. According to [41], we define COST of a strategy as the number of workers it needs to outperform the "Single Thread", which is a comprehensive measurement of both efficiency and scalability. In this experiment, we query q_1 and q_2 on the popular dataset LJ, and show the results in Figure 11. Note that we only plot the communication and memory consumption for q_1, as q_2 follows similar trend. We also test on the other datasets, such as the dense dataset GP, the results are also similar.

All strategies demonstrate reasonable scaling regarding both queries. In terms of COST, note that FullRep is slightly larger than 1, because "DualSim" is implemented in general for arbitrary query, while "SingleThread" uses a hand-tuned implementation. We first analyze the results of q_1. The COST ranking is FullRep (1.6), WOptJoin (2.0), BinJoin (3.1) and ShrCube (3.7). As expected, WOptJoin scales worse than FullRep, while BinJoin scales worse than WOptJoin because shuffling cost is increasing with the number of machines. In terms of memory consumption, it is trivial that FullRep constantly consumes memory of graph size. Due to the use of `Batching`, both BinJoin and WOptJoin consume very low memory for both queries. Observe that ShrCube consumes much more memory than WOptJoin and BinJoin, even more than the graph data itself. This is because that certain worker may receive more edges (with duplicates) than the graph itself, which increases the peak memory consumption. For communication cost, both BinJoin and WOptJoin demonstrate reasonable drops as the increment of machines. ShrCube renders much less communication as expected, but it shows increasing trend. This is actually a reasonable behavior of ShrCube, as more machines also means more data duplicates. For q_2, the COST ranking is FullRep (2.4), WOptJoin (2.75), BinJoin (3.82) and ShrCube (71.2). Here, ShrCube is dramatically larger, with most time spending on deduplication (Section 3.3). The trend of memory consumption and communication cost of q_2 is similar with that of q_1, thus is not further discussed.

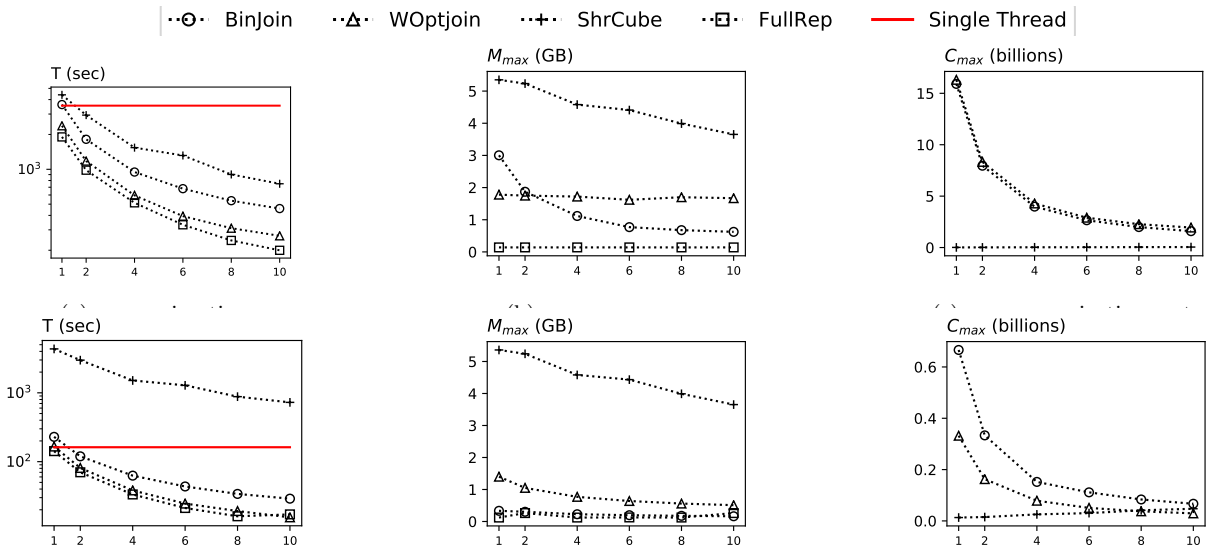
Exp-8 Vary Desities for Labelled Matching. Based on DG10, We generate the datasets with densities 10, 20, 40, 80 and 160 by randomly adding edges into DG10. Note that the density-10 dataset is the original DG10 in Table 5. We use the labelled queries q_4 and q_7 in this experiment, and show the results in Figure 12.

Exp-9 Vary Labels for Labelled Matching. We generate the datasets with number of labels 0, 5, 10, 15 and 20 based on DG10. Note that there are 5 labels in labelled queries q_4 and q_7, which are called the target labels. The 10-label dataset is the original DG10. For the one with 5 labels, we will replace each label not in the target labels as one random target label. For the ones with more than 10 labels, we randomly choose some nodes to change their labels into some other pre-defined labels until they contain the required number of labels. For the one with zero label, it degenerates into unlabelled matching, and we use unlabelled version of q_4 and q_7 instead. The experiment demonstrates the transition from unlabelled matching to labelled matching, where the biggest drop happens for all algorithms. The drops continue with the increment of the number of labels, but less sharply when there are sufficient number of labels (≥ 10). Observe that when there are very few labels, for example, the 5-label case of q_7, FullRep actually performs worse than BinJoin and WOptJoin. The "CFLMatch" algorithm [17] used by FullRep relies heavily on label-based pruning. Fewer labels render larger candidate set and more recursive calls, resulting in performance drop of FullRep. While fewer labels may enlarge the intermediate results of BinJoin and WOptJoin, but they are relatively small in the labelled case, and does not create much burden for the 10GBps network.

B. AUXILIARY MATERIALS

All Query Results. In Table 6, We show the number of results of every successful query on each dataset evaluated in this work. Note that DG10 and DG60 record the labelled queries of $q_1 - q_9$.

Join Plans. We show the detailed join plans of the queries we used in Exp-3 (Section 5). We do not present P_2 as it

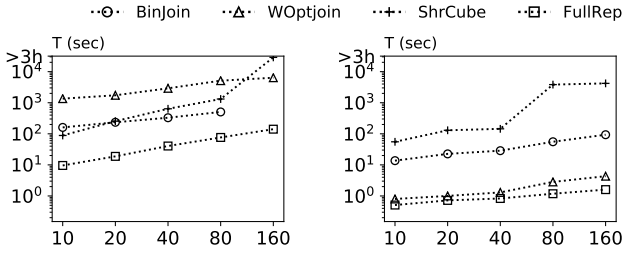


(d) q_2 running time (e) q_2 memory usage (f) q_2 communication cost

Figure 11: Scalability experiment: querying q_1 and q_2 on LJ.

Dataset	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9
GO	539.58M	621.18M	39.88M	38.20B	27.80B	9.28B	2,168.86B	330.68B	1.88T
GP	1.42T	1.16T	78.40B	-	-	-	-	-	-
US	1.61M	21,599	90	117,996	2,186	1	160.93M	2,891	89
LJ	51.52B	76.35B	9.93B	53.55T	44.78T	18.84T	-	-	-
UK	2.49T	2.73T	157.19B	-	-	-	-	-	-
EU	905,640	2,223	6	12,790	450	0	342.48M	436	71
FS	-	185.19B	8.96B	-	-	3.18T	-	-	-
SY	-	834.78B	5.47B	-	-	-	-	-	-
DG10*	40.14M	26.76M	28.73M	22.59M	23.08B	1.49M4	47,556	42.56M	10.07M
DG60*	302.41M	169.86M	267.38M	161.69M	203.33B	12.44M	983,370	4.14B	114.19M

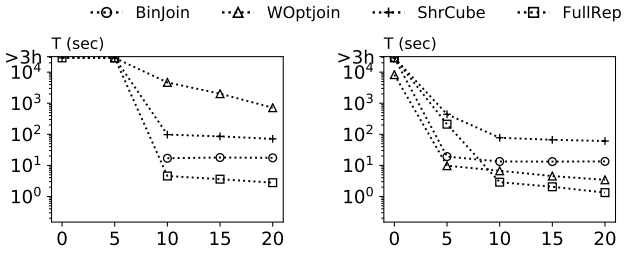
Table 6: All Query's Results.



(a) q_4 (b) q_7

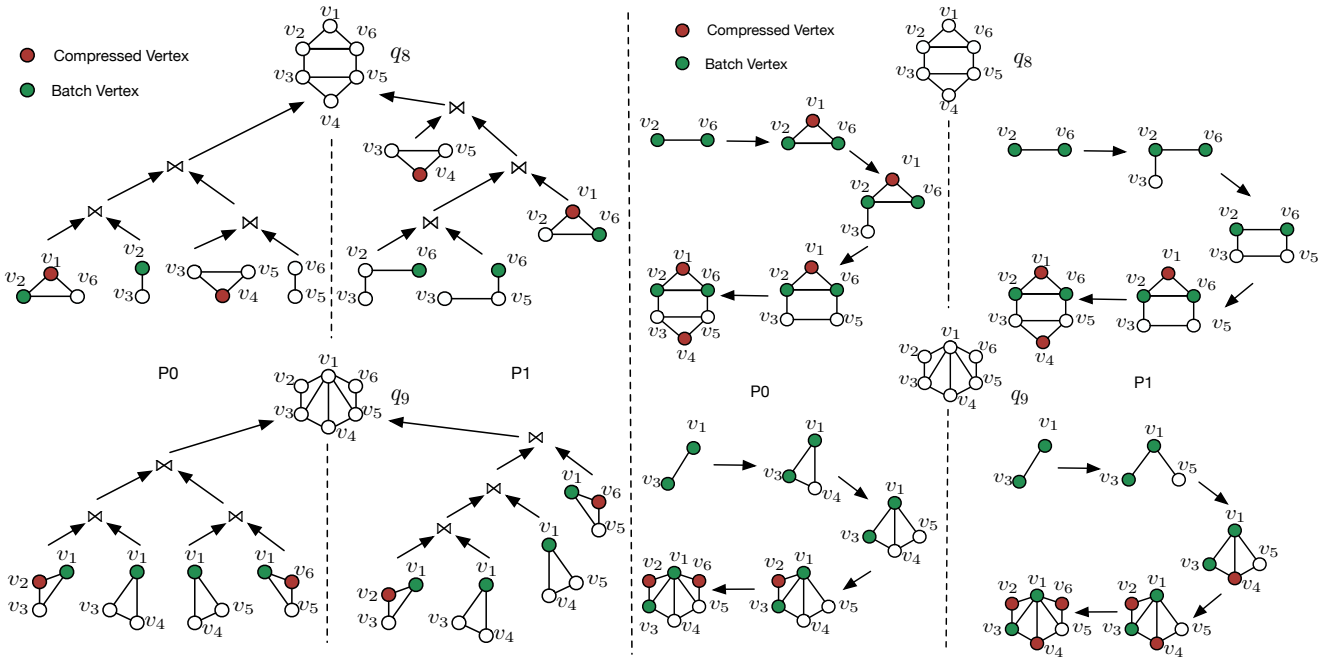
Figure 12: Varying densities of labelled graph.

is randomly generated, and gives poor performance in the experiment.



(a) q_4 (b) q_7

Figure 13: Varying labels of labelled graph.



(a) BinJoin (b) WOptJoin

Figure 14: The Join Plans of q_8 and q_9 in Exp-2.