Improving Distributed Subgraph Matching Algorithm on Timely Dataflow

Zhengmin Lai[1], Longbin Lai[2], and Zhengyi Yang[2]

[1]East China Normal University, Shanghai, China
[2]The University of New South Wales, Sydney, Australia
[1]zmlai@stu.ecnu.edu.cn
[2]{llai, zyang}@cse.unsw.edu.au

***Abstract*—The subgraph matching problem is a well-known NP-hard problem: given a query graph q and data graph G, we need to find all subgraphs matching q in G. Subgraph matching plays a vital role in the fields of e-commerce, social media and biological science. One of the most efficient and scalable distributed subgraph matching algorithm is CliqueJoin. While most of the graphs in real applications are labelled graphs, CliqueJoin does not propose a cost evaluation strategy for them, which limits its application in practice. In the meanwhile, CliqueJoin is based on Map-Reduce, by which the performance of the algorithm can be greatly affected due to the frequent I/O operations when handling intermediate results in the multi-round join process. Discovering the limitations of CliqueJoin, this paper propose GenCliqJoin by extending the cost evaluation function in CliqueJoin to generate optimal join plan for labelled graphs in the distributed context. What's more, this paper implements GenCliqJoin on Timely dataflow system instead of Map-Reduce to avoid frequent I/O operations. The generality and performance of CliqueJoin can be significantly improved after solving these two limitations. This paper also carries out large scale experiments, whose results show that the algorithm after being improved is 2 to 10 times faster than the original Map-Reduce version for unlabelled matching, and has excellent performance and scalability doing labelled matching.**

***Index Terms*—subgraph matching, cost evaluation, distributed algorithm, dataflow**

I. Introduction

In this paper, we study subgraph matching, which is one of the most fundamental problem about graph analysis. Given a query graph q and a data graph G, subgraph matching is defined to find all subgraph instances of G that are isomorphic to p. Subgraph matching is widely used in numerous domains. For example, it is used to illustrate the evolution process of social networks [22], identify terrorist cells in activity networks [12], and discover certain features of biological or chemical networks [9]. Subgraph matching is also a basic operation of the query language in graph databases such as Neo4j [16] and Gremlin [2].

Existing Solutions. Unfortunately, the computation of subgraph matching is proven to be NP-complete [31]. In recent years, a lot of effort has been contributed to improving the efficiency of subgraph matching. The naive approach is to perform brute-force search over all subgraphs of data graph G. Ullman's backtracking algorithm [36] studies on searching order, pruning rules and neighbourhood indexes to find all subgraphs. VF2 [13] and QuickSI [32] focus on designing better indexes or pruning rules to further improve the efficiency. However, these works are all sequential algorithms that can not handle large scale graphs in a single machine. Discovering the poor efficiency and scalability of sequential algorithms, researchers seek to develop subgraph matching algorithms in the distributed context. PSgL [33], StarJoin [34], TwinTwigJoin [24] and CliqueJoin [25] perform subgraph matching by joining the sub-structure of the query graph, where the basic sub-sturcture is called *join unit*. Those four algorithms mainly differ in the join order and join unit. PSgL is a variant of StarJoin. TwinTwigJoin is instance optimal to StarJoin. CliqueJoin extends StarJoin, where CliqueJoin can use both star and clique as join unit and its optimal join plan is expressed in the form of a bushy tree [21]. Instead of joining subgraphs, BigJoin [6] and CrystalJoin [28] adopt expanding vertices strategy in each round. Another algorithm MultiwayJoin [5] is to divide the search space evenly into workers, then each worker can finish the subgraph computation locally.

Motivation. CliqueJoin has been proven to be the most efficient algorithm in the category of joining subgraphs [25]. However, CliqueJoin has two limitations. First, it is implemented on Map-Reduce, where frequent I/O operations will occur in each round. Therefore, the performacne of the algorithm can be greatly affected. Second, while most of the graphs in real life are labelled graphs, it doesn't propose a cost evaluation strategy for them. Hence it can not generate optimal join plan for labelled graphs.

Contributions. In this paper, we propose GenCliqJoin by extending and improving the state-of-the-art algorithm CliqueJoin as follows:

(1) Generalizing CliqueJoin *to perform subgraph matching on labelled graphs.* We refine the cost evaluation function for CliqueJoin that can generate optimal join plan for labelled graphs. By doing this, we can easily extend CliqueJoin to do labelled subgraph matching. In consequence, the generality of CliqueJoin will be significantly improved.

(2) Reimplementing CliqueJoin *on* Timely *dataflow system*

[27]. We reimplement CliqueJoin on Timely dataflow system, a high performance distributed framework, to avoid frequent I/O operations in Map-Reduce. Hence the performance of CliqueJoin can be greatly improved.

(3) In-depth performance studies on large datasets. We perform large scale and in-depth experiments on both unlabelled and labelled graphs. Comparing with original version on Map-Reduce, our implementation speeds up CliqueJoin by 2 times to 10 times in general. The experimental results also show that GenCliqJoin has excellent performance and scalability doing labelled subgraph matching, even if the graph is of billion scale.

Organization. The rest of the paper is organized as follows. Section II introduces the definition of subgraph matching and preliminary knowledge. Section III introduces GenCliqJoin, which extends CliqueJoin to do labelled subgraph matching, and its implementation details on Timely dataflow system. Section IV illustrates the experimental results of doing subgraph matching on both unlabelled and labelled large graphs. Section V shows related works, and section VI concludes the paper.

II. Preliminaries

A graph g can be represented as $g = (V, E, L)$, where $V(g)$ is the vertex set of g, $E(g) \subseteq V(g) \times V(g)$ is the edge set of g, and L is a label function that maps each node $v \in V(g)$ and/or each edge $e \in E(g)$ to a label. For unlabelled graph, L simply maps all nodes and edges to $\emptyset$. For a node $u \in V(g)$, we use $id(u)$ to denote its index, $\mathcal{N}(u)$ to denote its neighbours, $d(u) = |\mathcal{N}(u)|$ to denote its degree, $N = |V(g)|$ to denote its node size, $M = |E(g)|$ to denote its edge size, $\overline{d(g)} = 2|E(g)|/|V(g)|$ and $D(g) = max_{u \in V(g)} d_g(u)$ to denote its average and maximum degree, respectively. A graph g is a *clique* if for any pair of nodes $u, v \in V(g)$, we have $(u, v) \in E(g)$. $k - clique$ is a clique consists of k nodes. A graph g is a *star* if it is a tree with depth 1. $k - star$ is a tree with one root node and k leaf nodes.

Given two graphs g_1 and g_2, g_1 is a subgraph of g_2 if $V(g_1) \subseteq V(g_2)$ and $E(g_1) \subseteq E(g_2)$.

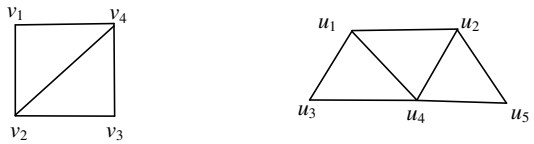
Definition 1: **(Subgraph Isomorphism)** Given a query graph q and data graph G, q is subgraph isomorphic to G iff there exists a bijective mapping function $f : V(q) \to V(G)$ such that, (1) $\forall v \in V(q)$, $L_q(v) = L_G(f(v))$; (2) $\forall (v_1, v_2) \in E(q)$, $(f(v_1), f(v_2)) \in E(G)$, and $L_q((v_1, v_2)) = L_G((f(v_1), f(v_2)))$.

Here, we call a subgraph isomorphism a *Match*. Given the query vertices $\{v_1, v_2, ..., v_n\}$, a match f can be represented as $\{u_{k_1}, u_{k_2}, ..., u_{k_n}\}$, where $u_{k_i} = f(v_i)$. An *automorphism* is a graph that is isomorphic to itself.

Problem Statement. Given a query graph q and data graph G, *subgraph matching* is to enumerate all subgraphs in G that are isomorphic to q.

Given a query graph q and data graph G, we denote the subgraph matching result set as $R_G(q)$(or $R(q)$ if the context is clear).

Example 1: In Figure 1, we give an unlabelled query graph q and data graph G. We use symmetry breaking technique [17] to assign partial order for query graph to avoid duplicated enumeration caused by automorphism. In this example, the partial order of query graph can be $\{v_1 < v_3, v_2 < v_4\}$. There are two matches: $\{u_2, u_1, u_3, u_4\}$ and $\{u_1, u_2, u_5, u_4\}$. We can check the order constraint for the first match, which is $\{u_2, u_1, u_3, u_4\}$. As we have order $\{v_1 < v_3, v_2 < v_4\}$, it constraints that we should have $f(v_1) < f(v_3)$, $f(v_2) < f(v_4)$,where $u_2 < u_3$, $u_1 < u_4$ satisfies this constraint. Note that we have $f(v_i) < f(v_j)$ iff $id(f(v_i)) < id(f(v_j))$.



(a) query graph (b) data graph

Fig. 1: An example of subgraph matching.

Regarding query vertices as attributes and data vertices as tuples in the relation table, we can naturally express the subgraph join process as joining relations. In Figure 1, the edge-by-edge join process can be demonstrated as

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

CliqueJoin. Generally speaking, the state-of-the-art algorithm CliqueJoin follows the decomposition-and-join framework to do subgraph matching. The main idea of CliqueJoin can be concluded as follows.

(1) SCP Storage Mechanism. In the SCP graph storage mechanism, denoted as $\Phi(G)$, we have $\Phi(G) = \{G_v \mid v \in V(G)\}$, $V(G_v) = v \cup \mathcal{N}(v)$ and $E(G_v) = \{(v, v') \mid v' \in \mathcal{N}(v)\} \cup \{(v', v") \mid v', v" \in \mathcal{N}(v) \wedge (v', v") \in E(G) \wedge v < v' \wedge v < v"\}$, where $G_v \subseteq G$ is a connected subgraph of G with $v \in V(G_v)$, and $\bigcup_{u \in V(G)} E(G_v) = E(G)$. Each G_v is called the *local graph* of v. Suppose the data graph G is maintained in the distributed file system in the form of key-value pairs $(v; G_v)$ for each $v \in V(G)$ according to $\Phi(G)$, a *join unit* is a structure whose matches can be enumerated independently in each local graph $G_v \in \Phi(G)$. For CliqueJoin, the join unit can either be a clique or a star.

(2) Query Decomposition. Given a graph storage $\Phi(G)$ and query graph q, a query decomposition of q is defined as $D = \{p_0, p_1, \ldots, p_t\}$, where each $p_i \in D (0 \leq i \leq t)$ is a *join unit* w.r.t. $\Phi(G)$ and $q = \bigcup_{p_i \in D} p_i$. Given the decomposition $D = \{p_0, p_1, \ldots, p_t\}$ of q, we solve the subgraph enumeration using t rounds of two-way join:

$$R(q) = \bowtie_{p_i \in D} R(p_i). \quad (2)$$

(3) Optimal Join Plan. A *join plan* determines an order to solve the above join, which can significantly affect the performance of the algorithm. The join plan is usually presented in a binary tree structure, where the leaf nodes are (the matches of) the join units, the internal nodes are the partial queries. Given a join plan, we compute the join order through post-order traversal [1] over its binary tree. We denote P as the partial queries set, P_i as the i-th partial query whose results are produced in the i-th round of the join plan. As a result, a join plan, denoted as J, can be uniquely represented as $J = (D, P)$. CliqueJoin utilizes general bushy tree [21] to represent its join plan.

Example 2: Figure 2 shows a join plan for an unlabelled query graph q in the form of a bushy tree. The decomposition of q is $D = \{p_0, p_1, p_2, p_3\}$, and partial query set is $P = \{P_1, P_2, P_3\}$, where $P_3 = q$. The first round of join is $P_1 = p_0 \bowtie p_1$, second round is $P_2 = p_2 \bowtie p_3$, and the final round is $P_3 = P_1 \bowtie P_2$. In this case, we use triangle (3-clique) as the join unit.

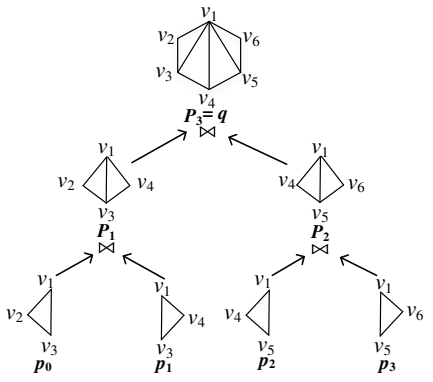


Fig. 2: An example of optimal bushy join plan.

We denote the set of all possible join plans for query graph q as $\mathcal{S}(q)$. Given a cost function $\mathcal{C}$ defined over $\mathcal{S}$, we say a join plan ε is *optimal* iff $\mathcal{C}(\varepsilon)$ is minimized. The details of $\mathcal{C}(\varepsilon)$ design can be found in [25]. Here, we only need to know that $\mathcal{C}(\varepsilon)$ is positive related to $|R(q)|$.

(4) Matching Result Estimation. In order to compute the join plan cost $\mathcal{C}(\varepsilon)$, CliqueJoin needs to estimate $|R(q)|$ for a given query graph q. As most of real-life graphs follow *power-law* random distribution [11], CliqueJoin estimates $|R_{G_{PR}}(q)|$ in data graph G_{PR} generated by power-law model.

Considering q is constructed from a single edge by extending one edge at a time in steps. Let $q^{(1)}$ and $q^{(2)}$ be two consecutive queries obtained along the process. More specifically, for some $v \in V(q^{(1)})$ and $v' \in V(q^{(2)})$ such that $(v, v') \notin E(q^{(1)})$, $q^{(2)}$ is obtained by adding the edge (v, v') to q^1. Suppose M is a match of $q^{(1)}$, in principal, we extend M by one more edge to get new matches for $q^{(2)}$. Thus, if the expectation of new matches that can be extended for one certain match of $q^{(1)}$ is λ, we have:

$$|R_{G_{PR}}(q^{(2)})| = \lambda |R_{G_{PR}}(q^{(1)})| \quad (3)$$

The value of λ depends on the edge which extends $q^{(1)}$ to $q^{(2)}$. There are two cases may happen, that is, $v' \notin V(q^{(1)})$ and $v' \in V(q^{(1)})$. The details of the two cases for computing λ and the algorithm of computing $|R_{G_{PR}}(q)|$ by extending edges can be found in [25].

Timely Dataflow. Timely dataflow system is a high performance distributed system. It abstracts the computation model as a dataflow graph. The node in the dataflow graph is responsible for doing computations and the edge is to send data streams to nodes. One node can receive several input streams and produce one output stream. When the dataflow graph for the given computation task is constructed, it will distribute data to each worker in the cluster, and each worker can finish its computation locally. The whole computation task is finished when there is no output stream produced by workers.

III. Optimization

CliqueJoin is proposed for unlabelled graphs and implemented in Map-Reduce. In this section, we introduce GenCliqJoin, our revision of CliqueJoin to extend the algorithm to labelled graphs and dataflow model.

A. Cost Analysis for labelled Matching

We can use the label information in the graph to refine the result size estimation strategy in GenCliqJoin. Intuitively, we should process query graphs with rare labels as fast as possible to reduce the cost.

We use $\Pr_G(\iota)$ to denote the probability of the label ι that appears in a certain node of data graph G. Then, (3) can be refined as:

$$|R_G(q^{(2)})| = \eta\lambda |R_G(q^{(1)})| \quad (4)$$

Similarly, the value of η is considered in two cases:

- **(Case 1)** If $v' \notin V(q^{(1)})$, a new vertex is introduced in $q^{(2)}$ as well as a new edge. In this case, we have:

$$\eta = \Pr_G(l(v')) \times \Pr_G(l((v, v'))) \quad (5)$$

- **(Case 2)** If $v' \in V(q^{(1)})$, an edge is added between two existing vertices in $q^{(1)}$. In this case, we have:

$$\eta = \Pr_G(l((v, v'))) \quad (6)$$

In addition, $\Pr_G(\iota)$ is calculated using the maximum likelihood estimation via sampling or scanning the data graph G.

B. Migration from MapReduce to Dataflow

Implementation Details. The SCP storage mechanism and optimal join plan generation(support both labelled and unlabelled queries) are strictly implemented according to [25] and the optimized cost model we propose in Section III. Here, given a join plan J, we will show the details of how to implement GenCliqJoin in Timely dataflow.

(1) Building Timely *Dataflow.* The building Timely dataflow procedure is shown in Algorithm 1. This algorithm is to compute the join operations round by round to get the final matches $R(q)$ in stream.

There are two inputs of the algorithm: a $InputHandle$ set I and join plan J. An $InputHandle$ is a handler in Timely to store data. The data in $InputHandle$ can be converted to stream directly by invoking to_stream operation in Timely. We use I to store all join unit's matches $R(p_i)$, where $p_i \in D$. Join plan J consists of $|J|$ round join configurations. One round configuration $j \in J$ includes left subgraph, denoted as $j.lg$, right subgraph, denoted as $j.rg$, join key, denoted as $j.join\_key$, and batch parameters, denoted as $j.batch\_params$. With the configurations in j, we know exactly what we should do in each round of join.

In line 1-2, if there is only one element in I, it means the query graph q is a join unit, and we don't need to do join operations. Therefore, we just return the streaming result of $I[0]$. In line 3, for the join order is consistent with post-order traversal over the binary tree(e.g. Figure 2), we use a stack *StreamStack* to store $R(P_i)$, where P_i is a partial query. In line 5, for each join j in J, we first find its left and right subgraph matches in stream, denoted as $LStream, RStream$(line 6), respectively. In line 6-16, we consider two cases: (1) If the left/right subgraph is a join unit, we simply fetch its stream in I. (2) If left/right subgraph is a partial query, we can get its stream by popping the top element in $StreamStack$. Then we use `BatchJoin` to join $LStream$ and $RStream$ under the configuration j(line 17), and push the result stream $RstStream$ onto $StreamStack$(line 18). When we complete the iteration over J, we pop the top element as the final result stream, which is $R(q)$.

(2) Computing Join Unit Matches. Before we run Algorithm 1, we need to precompute all join unit matches $R(p_i)$, where $p_i \in D$. The process of computing $R(p_i)$ is illustrated in Algorithm 2. The inputs of the algorithm include join plan J and the local data graph G. In line 1, we initialize *InputHandle* set I with length $|J.D|$, which is exactly the number of join units in J. In line 3-11, for each join $j \in J$, if left/right subgraph $j.lg/j.rg$ is a join unit, we compute its matches $R_G(j.lg)/R_G(j.rg)$ in local graph G and send them to the corresponding input handler $I[i]$, where p_i is the i-th join unit in J. In line 12, we return the join unit matches I.

(3) Joining Two Streams in Batch. We observe that when directly implementing join of two streams, the huge intermediate results on large data graph will consume up the memory. Therefore, we implement the external hash join following *buffer-and-batch* idea to save memory, which is shown in Algorithm 3. The inputs of the algorithm are two streams S_1, S_2 and the join configuration j. In line 1-2, we buffer the data in S_1/S_2 to B_1/B_2. More specifically, we buffer the data from stream to a given threshold configured in $j.batch\_params$, and sort it according to $j.join\_key$ before spilling to the disk. In line 3-7, while the buffered data B_1/B_2 is not empty, we read the data from disk to $buffer_1/buffer_2$ batch by batch(line 4-5). Then we join $buffer_1$ and $buffer_2$ according to the join key in j, and output the result(line 6-7). In this way, the memory consumed by each join is one batch of data and we can configure the batch size in $j.batch\_params$ according to the machine's memory capacity.

Algorithm 1: `BuildDataflow`($InputHandle$ set I, join plan J)

```
1  if |I| = 1 then
2      Return I[0].to_stream();
3  StreamStack ← ∅;
4  i ← 0;
5  forall j ∈ J do
6      LStream ← ∅; RStream ← ∅;
7      if j.lg is a join unit then
8          LStream = I[i].to_stream();
9          i ← i + 1;
10     else
11         LStream = StreamStack.Pop();
12     if j.rg is a join unit then
13         RStream = I[i].to_stream();
14         i ← i + 1;
15     else
16         RStream = StreamStack.Pop();
17     RstStream = BatchJoin(LStream, RStream, j);
18     StreamStack.Push(RstStream);
19 Return StreamStack.Pop();
```

Algorithm 2: `CompUnitMatches`(join plan configuration set J, local data graph G)

```
1  I ← new InputHandle set with size of |J.D|;
2  i ← 0;
3  forall j ∈ J do
4      if j.lg is a join unit then
5          M ← R_G(j.lg);
6          send M to I[i];
7          i ← i + 1;
8      if j.rg is a join unit then
9          M ← R_G(j.rg);
10         send M to I[i];
11         i ← i + 1;
12 Return I;
```

Algorithm 3: `BatchJoin`(left stream S_1, right stream S_2, join configuration j)

```
1  B_1 ← S_1.buffer(j);
2  B_2 ← S_2.buffer(j);
3  while B_1 ≠ ∅ and B_2 ≠ ∅ do
4      buffer_1 ← B_1.next_batch();
5      buffer_2 ← B_2.next_batch();
6      result ← buffer_1 ⋈ buffer_2 according to j.join_key;
7      Output result;
```

Datasets	Name	$\|V(G)\|/mil$	$\|E(G)\|/mil$	$\overline{d(G)}$	$D(G)$
livejournal	LJ	4.85	43.37	8.9	20,333
orkut	OK	3.07	117.19	38.1	33,313
uk2002	UK	18.5	298.11	16.1	194,955

TABLE I: The Unlabelled Datasets.

IV. Experiments

In this section, we conduct extensive experiments for GenCliqJoin on both unlabelled and labelled data graphs. Here, we only consider undirected graph matching because an undirected edge can be regarded as two directed edges. We will show the experimental results over the optimizations we have done to CliqueJoin.

A. Experimental Settings

Environments. We use a cluster of 10 nodes connected via a 10Gpbs switch, and each node has 64GB memory, 1TB disk and 1 Intel Xeon CPU E3-1220 V6 3.00GHz with 4 physical cores. We implement CliqueJoin in Timely dataflow system [1] using Rust 1.27. We use 10 machines and each machine uses 3 workers by default.

Metrics. We measure query time T from the average time of three runs. The query time is actually the slowest worker's running time during each run, which excludes graph loading time as it is negligible compared to query time. We set batch size to 10,000 and threshold to 10,000,000 by default in `BatchJoin`.

Preprocessing Datasets. We preprocess each dataset as follows: we treat it as a simple undirected graph by removing self-loop and duplicate edges, and reorder the node id according to the degree and represent it using Compressed Sparse Raw (CSR)[2].

B. Unlabelled Matching

Datasets. We evaluate 3 real graphs of different sizes and types. The sources of these graphs are from Snap[3] and WebG[4]. We consider the following two graph types: Social Network (Soc) and Web Graph (Web). We list the evaluated graphs in Table I. Note that both $|V(G)|$ and $|E(G)|$ are in millions.

Queries. The five queries denoted by q_1 to q_5 are illustrated in Figure 3, where the number of nodes vary from 4 to 6 and the number of edges vary from 4 to 10. We assign the order of the nodes for symmetry breaking [17] under each query graph. Here, we have considered all queries for fair comparisons.

Exp1 - Vary Queries. We compare GenCliqJoin with CliqueJoin by testing all queries on LJ, which is shown in Figure 4. We can see that for enumerating the join unit q_3(4-clique) and q_5(5-clique), GenCliqJoin outperforms CliqueJoin by more than one order of magnitude. For q_2, GenCliqJoin is 2x faster than CliqueJoin. However, CliqueJoin outperforms GenCliqJoin in q_1 and q_5. TODO: Explain them.

[1]https://github.com/frankmcsherry/timely-dataflow
[2]https://en.wikipedia.org/wiki/Sparse_matrix
[3]http://snap.stanford.edu/data/index.html
[4]http://law.di.unimi.it/datasets.php

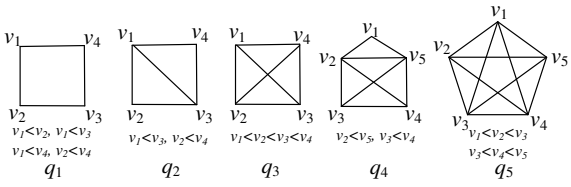


Fig. 3: Unlabelled Queries.

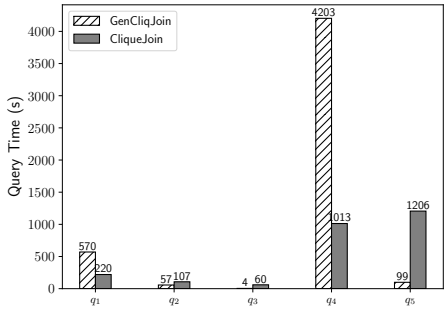


Fig. 4: Vary Queries.

Exp2 - Vary Datasets. We compare GenCliqJoin with CliqueJoin by querying q_2 and q_5 on all datasets in order to show the good performance over different data properties. The results are shown in Figure 5. We can see that, for querying q_2, GenCliqJoin generally outperforms CliqueJoin by around 2 times, despite of using fewer workers compared with CliqueJoin's workers in cluster. When querying q_5, GenCliqJoin is 3 to 10 times faster than CliqueJoin.

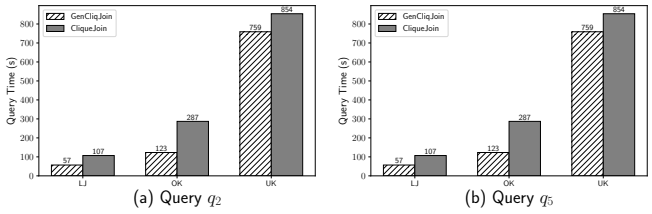


Fig. 5: Vary Datasets.

Exp3 - Scalability. We compare the scalability of GenCliqJoin with CliqueJoin on LJ using q_2 by varying number of nodes(6, 8, 10) used in the cluster, whose results are shown in Figure 6. We can see that GenCliqJoin is in general 2 times faster than CliqueJoin.

C. Labelled Matching.

We use LDBC social network benchmarking (SNB) [3] for labelled matching experiment for the lack of big public labelled graphs. SNB provides a data generator that can

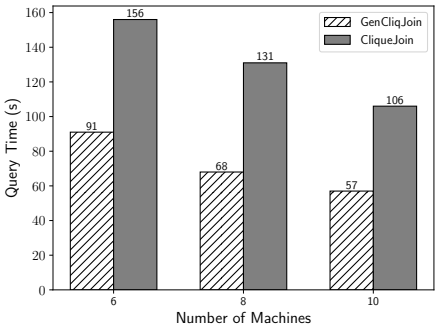


Fig. 6: Unlabelled Scalability.

Name	$\|V(G)\|/mil$	$\|E(G)\|/mil$	$\overline{d(G)}$	$D(G)$	#Labels
DG01	3.2	17.24	10.84	464,368	11
DG03	9.28	52.65	11.3	1,346,287	11
DG10	29.99	176.48	11.77	4,282,812	11
DG30	88.79	540.51	12.17	12,684,488	11
DG60	187.11	1246.66	13.32	26,639,563	11

TABLE II: The Labelled Datasets.

generate synthetic social networks of statistics, and a document [4] that describes the benchmarking tasks, which are actually doing subgraph matching. To the best of our knowledge, there are few experiments of distributed labelled subgraph matching on large datasets. Therefore, in this section, we will just demonstrate the effectiveness and scalability of GenCliqJoin when doing labelled subgraph matching.

Datasets. We list the datasets and their statistics in Table II. All datasets are generated using the "Facebook" mode with a span of 3 years. The dataset's name, denoted as DGx, represents a scale factor of x. As mentioned before, we first parse the graph into undirected simple graph. Then we remove all properties except the node types as labels, and the label is encoded as an integer to accelerate the matching.

Queries. The labelled queries are shown in Figure 7, which are generated from SNB's tasks with following rules: (1) removing the direction of edges and edge labels; (2) using one-hop edge for multi-hop edges; (3) removing the "no edge" and unconnected graph condition; (4) removing all properties except the node type as its label. For (1), we do the adaptation for simplicity although we can support that case. We adapt (2) and (3) for consistency with the subgraph matching problem studied in this paper. We adapt (4) for our implementation currently can not support property graphs.

Exp4 - All Labelled Queries. We perform GenCliqJoin for all queries on all datasets, and the results are illustrated in Figure 8. We can see that GenCliqJoin can finish subgraph matching in tens of seconds for q_2, q_3, q_4, q_5, q_6 in all data graphs, even if DG60 is a billion scale graph. We notice that the query time for q_1 increases sharply when the dataset becomes larger. The reason is that the algorithm spends a lot of time computing the stars' matches in q_1(q_1 is decomposed

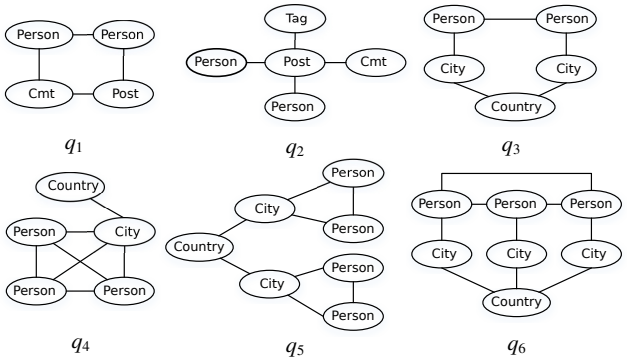


Fig. 7: Labelled Queries.

to two 2-star(s)) due to the poor filter information of q_1's join units in data graph.

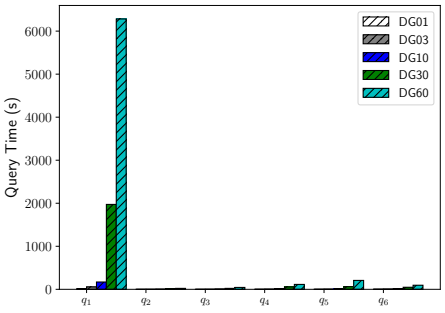


Fig. 8: All Labelled Queries.

Exp5 - Labelled Scalability. We evaluate the labelled matching scalability of q_1 and q_4 on DG10 by varying the number of nodes used in the cluster(6, 8, 10), and the results are demonstrated in Figure 9. We can see that when decreasing the number of machines used in the cluster, the query time slightly increases, which show that GenCliqJoin has great scalability for labelled matching.

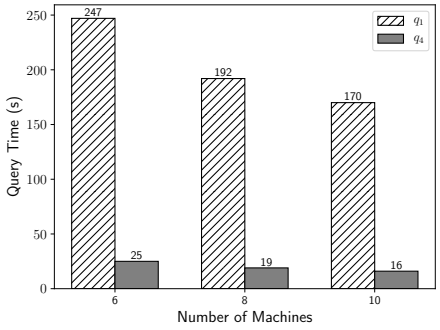


Fig. 9: Labelled Scalability.

V. Realted Work

Subgraph Enumeration. Besides the sequential algorithms we have mentioned in Section I, GraphQL [20] and SPath [38] focus on reducing the candidates of query vertices by exploiting neighborhood-based filtering. TurboISO [18] and the boost technique in [29] propose to merge vertices with same labels and the same neighbours in q and G respectively to reduce the matching complexity. [26] provides an in-depth comparison of above mentioned subgraph isomorphism algorithms. A more recent work in [7] uses a data structure called *compact path index* (CPI) to store the potential embeddings of a spanning tree of the query graph to improve both time and space efficiency. Algorithms of subgraph enumeration mainly focuses on answering a single query, [30] studied the problem of *multiple query optimization* (MQO) for subgraph enumeration. The details of distributed subgraph enumeration algorithms can be found in Section I.

Subgraph Containment Search. Let $\mathcal{D} = \{g_1, g_2, \ldots, g_N\}$ be a graph database that has N graphs, the problem of subgraph containment search over a graph database is to identify if the graphs in $\mathcal{D}$ contain the given query graph q. To speed up the search, many graph-feature based approaches have been proposed, performing graph indexing and adopting a filter-and-verification framework. As a result of such approach, false positives are removed by a pruning strategy before subgraph isomorphism algorithm is performed on each of the remaining candidates to obtain the final results. Existing works includes *frequent subgraph mining based approaches* (e.g., gIndex [37], Tree+Δ [39], and FG-Index [10]) and *exhaustive enumeration based approaches* (e.g., gCode [40], CT-Index [23], GraphGrep [15], GraphGrepSX [8], Closure-tree [19], and Grapes [14]). In approximate graph containment search, TALE [35] was proposed.

VI. Concolusion

In this paper, we study the distributed subgraph matching algorithm CliqueJoin. Discovering the limitations of CliqueJoin that can not handle labelled subgraph matching, we propose GenCliqJoin to generalize it by extending its cost evaluation function to labelled graphs so that it can generate optimal join plan for labelled query graphs. We further improve its performance by migrating CliqueJoin from Map-Reduce to Timely dataflow system, which can significantly reduce the I/O cost. We conduct extensive experiments on both unlabelled and labelled matching. The experimental results show that the generality and performance of CliqueJoin are highly improved after performing the two optimizations we propose in this paper.

References

[1] Data stucture algorithms. https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm.

[2] Gremlin. https://tinkerpop.apache.org/gremlin.html.

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

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

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

[6] 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.

[7] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214. ACM, 2016.

[8] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In *IAPR International Conference on Pattern Recognition in Bioinformatics*, pages 195–203. Springer, 2010.

[9] M. Cannataro, P. H. Guzzi, and P. Veltri. Protein-to-protein interactions: Technologies, databases, and algorithms. *ACM Comput. Surv.*, 43(1):1:1–1:36, Dec. 2010.

[10] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 857–872. ACM, 2007.

[11] F. Chung, L. Lu, and V. Vu. Eigenvalues of random power law graphs. *Annals of Combinatorics*, 7(1):21–33, Jun 2003.

[12] D. J. Cook and L. B. Holder. *Mining graph data*. John Wiley & Sons, 2006.

[13] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[14] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS one*, 8(10):e76911, 2013.

[15] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 2, pages 112–115. IEEE, 2002.

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

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

[18] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 337–348. ACM, 2013.

[19] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 38–38. IEEE, 2006.

[20] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418. ACM, 2008.

[21] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. *ACM SIGMOD Record*, 20(2):168–177, 1991.

[22] S. R. Kairam, D. J. Wang, and J. Leskovec. The life and death of online groups: Predicting group growth and longevity. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 673–682. ACM, 2012.

[23] K. Klein, N. Kriege, and P. Mutzel. Ct-index: Fingerprint-based graph indexing combining cycles and trees. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1115–1126. IEEE, 2011.

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

[25] 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.

[26] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the VLDB Endowment*, volume 6, pages 133–144. VLDB Endowment, 2012.

[27] 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.

[28] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment*, 11(2):176–188, 2017.

[29] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment*, 8(5):617–628, 2015.

[30] X. Ren and J. Wang. Multi-query optimization for subgraph isomorphism search. *Proceedings of the VLDB Endowment*, 10(3):121–132, 2016.

[31] 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.

[32] 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.

[33] 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.

[34] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9), 2012.

[35] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 963–972. IEEE, 2008.

[36] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[37] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 335–346. ACM, 2004.

[38] P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.

[39] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree+ delta¡= graph. In *Proceedings of the 33rd international conference on Very large data bases*, pages 938–949. VLDB Endowment, 2007.

[40] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 181–192. ACM, 2008.