Improving Distributed Subgraph Matching Algorithm on Timely Dataflow

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

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

***Abstract*—The subgraph matching problem is defined to find all subgraphs of a data graph that are isomorphic to a given query graph. Subgraph matching plays a vital role in the fields of e-commerce, social media and biological science. CliqueJoin is a distributed subgraph matching algorithm that is designed to be efficient and scalable. However, CliqueJoin is originally developed on MapReduce, thus the performance of the algorithm may be affected by the notorious I/O issue of MapReduce while processing multi-round task. Meanwhile, CliqueJoin does not propose a cost evaluation strategy for labelled graphs, which limits its application in practice where most graphs are labelled. Targeting the limitations of CliqueJoin, we propose CliqueJoin++ to improve CliqueJoin in two aspects. Firstly, we implement CliqueJoin on Timely dataflow system instead of MapReduce to avoid the I/O issue. Secondly, we extend the cost evaluation function in CliqueJoin to compute optimal join plan for labelled graphs in the distributed context. We conduct extensive experiments to show that the proposed method is up to 10 times faster than the MapReduce version for unlabelled matching, and it achieves good performance and scalability for labelled matching.**

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

I. Introduction

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 q. Subgraph matching is one of the most fundamental problem about graph analysis, and is widely used in many domains. For example, it is used to illustrate the evolution process of social networks [23], to identify terrorist cells in activity networks [13], and to discover certain features of biological or chemical networks [9]. Subgraph matching is also a basic operation of graph databases such as Neo4j [17] and graph languages such as Gremlin [2].

Existing Solutions. Despite its usefulness, subgraph matching is computationally intensive because it depends on the NP-complete operation, *subgraph isomorphism* [33]. In recent years, people have devoted a lot of efforts to improving the efficiency of subgraph matching. The first practical algorithm for computing subgraph isomorphism problem was proposed by Ullmann [38]. VF2 [14], QuickSI [34], TurboISO [19] and CFL [7] focus on designing better pruning rules and/or indexing structures to further improve the performance. However, these works are all sequential algorithms that can hardly handle large-scale graphs. While finding out the natural binding between subgraph matching and relational joins, people turned to develop both efficient and scalable distributed algorithms for subgraph matching by processing join operations. StarJoin [36] is the first algorithm to tackle subgraph matching via join in the distributed context. The idea is to decompose the query graph into a set of stars (star is a tree of depth 1), computing matches of each star and then joining these star matches to form the final result. This idea, called *decomposition-and-join*, inspires a series of following-up works including PSgL [35], TwinTwigJoin [25] and CliqueJoin [26]. CliqueJoin develops the state-of-the-art solution in the *decomposition-and-join* scheme, which decomposes the query into both stars and cliques (a complete graph), and then processes the **optimal** bushy join. There are some alternative schemes other than decomposition-and-join. Ammar et al. proposed BigJoin that solves the join using a variance of worst-case optimal join algorithm [30]. Afrati et al. proposed MultiwayJoin [5] that attempts to divide the search space evenly into the configured workers in the cluster, then each worker can process its own computation locally.

Motivations and challenges. CliqueJoin is the optimal algorithm in the decomposition-and-join scheme [26]. However, CliqueJoin has two limitations. First, it is implemented on MapReduce, which requires the data to be written to the disk at the end of certain round and read back from disk for the consecutive round. This can cause severe bottleneck while processing multi-round I/O-intensive task such as subgraph matching. Second, while most of the graphs in real life are labelled graphs, CliqueJoin only considers unlabelled matching, and may not compute optimal join plan for labelled matching without considering the labels' statistics.

In this paper, we propose CliqueJoin++ to resolve the issues of CliqueJoin. To avoid the I/O issue of MapReduce, we implement the algorithm on the Timely dataflow system [29]. Timely is a more general distributed engine than MapReduce. Instead of constraining the data to be written to the disk at the end of a computing round, Timely allows the user to program the desired way of maintaining the output data (internal memory by default). To extend CliqueJoin's usability in labelled matching, we refine the cost evaluation function for CliqueJoin by taking the labels' frequencies into consideration in order to compute optimal join plan for labelled graphs.

The implementation of CliqueJoin++ is non-trivial. Firstly, we target a generic implementation that can solve any query. Note that the original implementation of CliqueJoin is hard-coded for each query, which largely eases the implementation. It is challenging to translate the join plan of an ambiguous query into Timely-executable dataflow. Secondly, the cost function for unlabelled matching in CliqueJoin is already complicated, while we still need to inject the labels' statistics into the cost function for better estimation of the labelled matching.

There are some alternatives of optimizing distributed subgraph matching.

1) Alternative computing engines. There are alternative engines such as Spark [40] and Flink [10] to replace MapReduce. We tend to use Timely here due to (1) it is more flexible to program; (2) it draws lower system cost compared to the other popular engines [28].
2) Alternative algorithms. There are at least two alternative algorithms to study, namely BigJoin and MultiwayJoin. However, they do not draw our attention due to the following reasons. BigJoin does not invent a new join scheme, but simply adopts different algorithm to process the join. We can also implement the join scheme of CliqueJoin using BigJoin's join algorithm, which is also suggested by the authors of BigJoin [6]. Therefore, it is more fundamental to study CliqueJoin. Although MultiwayJoin does follow a different join scheme, researchers have already observed that it can end up with carrying the whole graph in each worker and thus can scale out poorly [6], [26].

Contributions. In this paper, we propose CliqueJoin++ by extending and improving the state-of-the-art algorithm CliqueJoin. We make the following contributions.

(1) Reimplementing CliqueJoin *on* Timely *dataflow system* [29]. We reimplement CliqueJoin for any generic query on Timely dataflow system, a high throughput distributed data engine, to avoid the I/O issue of Map-Reduce.

(2) Generalizing CliqueJoin *for labelled subgraph matching.* We refine the cost evaluation function for CliqueJoin that can generate optimal join plan for labelled graphs. As most of real graphs are labelled, we greatly expand the usability of CliqueJoin in practice.

(3) In-depth performance studies on large datasets. We perform in-depth experiments on both unlabelled and labelled graphs. For unlabelled experiment, we observe that our implementation can speed up CliqueJoin by up to 10 times. For labelled experiment, the results demonstrate that CliqueJoin++ achieves good performance and scalability, and can process data graph of billions of edges.

Organization. The rest of the paper is organized as follows. Section II introduces the definition of subgraph matching and preliminary knowledge. Section III introduces CliqueJoin++ , 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 graphs. Section V shows related works, and section VI concludes the paper.

II. Preliminaries

A graph g is represented as a tuple $g = (V, E, \Sigma_V, \Sigma_E, L)$, where $V(g)$ is the vertex set of g, $E(g) \subseteq V(g) \times V(g)$ is the edge set of g, Σ_V and Σ_E are the sets of vertex and edge labels, and L is a label function that maps each node $v \in V(g)$ and each edge $e \in E(g)$ to a label. For unlabelled graph, Σ_V and Σ_E are $\emptyset$. For a node $v \in V(g)$, we use $id(v)$ to denote its index, $Nbr(v)$ to denote its neighbours, $d(v) = |Nbr(v)|$ to denote its degree, $N = |V(g)|$ and $M = |E(g)|$ to denote its node and edge size, respectively. We use $d_{avg}(g) = 2N/M$ and $d_{max}(g) = MAX_{v \in V(g)} d_g(v)$ to denote g's average and maximum degree, respectively. A graph g is a *clique* if it is a complete graph. $k - clique$ is a clique consisting 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, denoted as $g_1 \subseteq g_2$ iff $V(g_1) \subseteq V(g_2)$ and $E(g_1) \subseteq E(g_2)$.

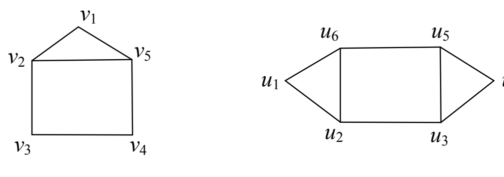
Definition 1: **(Subgraph Isomorphism)** Given a query graph q and data graph G, q is subgraph isomorphic to G iff there exists a bijective mapping function $f : V(q) \to V(G)$ such that, (1) $\forall v \in V(q)$, $L_q(v) = L_G(f(v))$; (2) $\forall (v_1, v_2) \in E(q)$, $(f(v_1), f(v_2)) \in E(G)$, and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

Here, we call a subgraph isomorphism f a *Match*, which can be represented as a tuple Tu consisting of data vertices and each data vertex u_j in Tu matches the vertex v_i in query graph($u_j = f(v_i)$). 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 [18] 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 $\{v_2 < v_5\}$. There are two matches of $(v_1, v_2, v_3, v_4, v_5)$, which are $(u_1, u_2, u_3, u_5, u_6)$ and $(u_4, u_3, u_2, u_6, u_5)$. We can check the order constraint for the first match, which is $(u_1, u_2, u_3, u_5, u_6)$. As we have order $v_2 < v_5$, it constraints that we should have $f(v_2) < f(v_5)$, where $u_2 = f(v_2), u_6 = f(v_5)$ and $u_2 < u_6$ satisfies this constraint. Note that for two nodes $u_i, u_j \in V(G)$, we have $u_i < u_j$ if $id(u_i) < id(u_j)$.



(a) query graph q (b) data graph G

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

$$R(q) = R(v_1, v_2) \bowtie R(v_2, v_3) \bowtie R(v_3, v_4) \bowtie R(v_4, v_5) \bowtie R(v_1, v_5) \bowtie R(v_2, v_5). \quad (1)$$

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 storage mechanism for data graph G, denoted as $\Phi(G)$, we have $\Phi(G) = \{G_v \mid v \in V(G)\}$, $V(G_v) = \{v\} \cup Nbr(v)$ and $E(G_v) = \{(v, v') \mid v' \in Nbr(v)\} \cup \{(v', v") \mid v', v" \in Nbr(v) \wedge (v', v") \in E(G) \wedge v < v' \wedge v < v"\}$, 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 [22] 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.

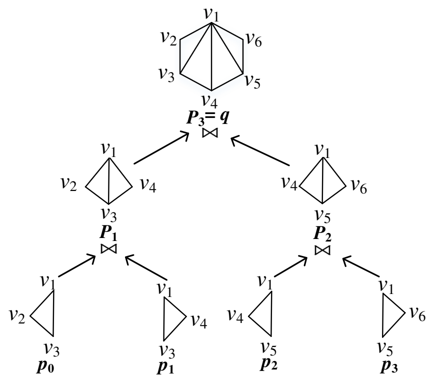


Fig. 2: An example of optimal bushy join plan.

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 [26]. 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 [12], 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 f is a match of $q^{(1)}$, in principal, we extend f 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 [26].

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. Optimizations

CliqueJoin is proposed for unlabelled graphs and implemented on MapReduce. In this section, we introduce CliqueJoin++ , 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 CliqueJoin++ . 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:

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

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:

$$\eta = \Pr_G(L(v')) \times \Pr_G(L((v, v'))) \tag{5}$$

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

$$\eta = \Pr_G(L((v, v'))) \tag{6}$$

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 [26] 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 CliqueJoin++ on 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 $R(q)$.

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

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

(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.

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

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

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

```
1  $B_1 \leftarrow S_1.buffer(j)$;
2  $B_2 \leftarrow S_2.buffer(j)$;
3  while $B_1 \neq \emptyset$ and $B_2 \neq \emptyset$ do
4      $buffer_1 \leftarrow B_1.next\_batch()$;
5      $buffer_2 \leftarrow B_2.next\_batch()$;
6      $result \leftarrow buffer_1 \bowtie buffer_2$ according to $j.join\_key$;
7      Output $result$;
```

IV. Experiments

In this section, we conduct extensive experiments for CliqueJoin++ 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.

Datasets	Name	N/mil	M/mil	$d_{avg}(G)$	$d_{max}(G)$
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.

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 $N = |V(G)|$ and $M = |E(G)|$ 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 5 and the number of edges vary from 4 to 10. We assign the order of the nodes for symmetry breaking [18] under each query graph. Here, we have considered all queries for fair comparisons.

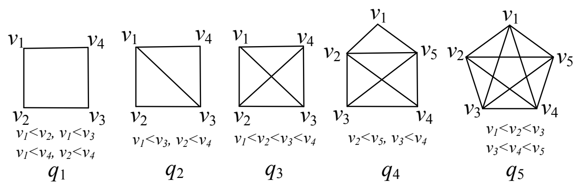


Fig. 3: Unlabelled Queries.

Exp1 - Vary Queries. We compare CliqueJoin++ with CliqueJoin by testing all queries on LJ, which is shown in Figure 4. Note that for better presentation, the ordinate is calculated by $logT$, where T is the query time. We can see that for enumerating the join unit q_3(4-clique) and q_5(5-clique), CliqueJoin++ outperforms CliqueJoin by more than 10

[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

times. For q_2, CliqueJoin++ is 2 times faster than CliqueJoin. However, CliqueJoin outperforms CliqueJoin++ in q_1 and q_4. The reason is that the original implementation of CliqueJoin is hard-coded for each query, where it can do specific optimizations for a certain query. However, since CliqueJoin++ targets a generic implementation that can handle any query, we can not perform many optimizations for queries, which may cause incompetent results compared with CliqueJoin in some queries like q_1 and q_4.

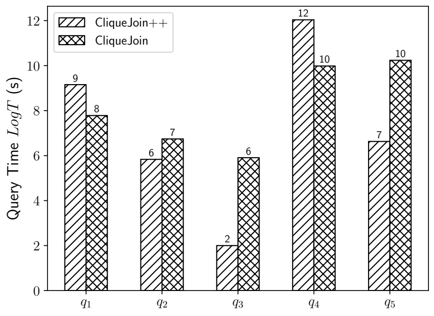


Fig. 4: Vary Queries.

Exp2 - Vary Datasets. We compare CliqueJoin++ 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, when querying q_2, CliqueJoin++ generally outperforms CliqueJoin by around 2 times. When querying q_5, CliqueJoin++ is 3 to 10 times faster than CliqueJoin.

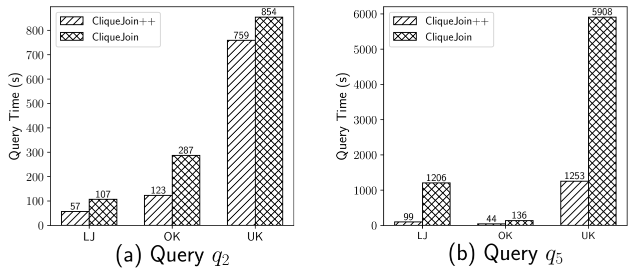


Fig. 5: Vary Datasets.

Exp3 - Scalability. We compare the scalability of CliqueJoin++ 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 CliqueJoin++ 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 generate synthetic social networks of statistics, and a document

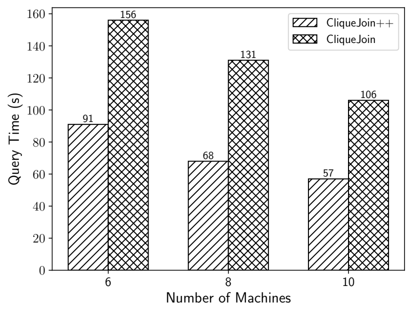


Fig. 6: Unlabelled Scalability.

Name	N/mil	M/mil	$d_{avg}(G)$	$d_{max}(G)$	#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.

[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 CliqueJoin++ 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 CliqueJoin++ for all queries on all datasets, and the results are illustrated in Figure 8(whose ordinate is the logarithm of query time T). We can see that CliqueJoin++ 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 to two 2-star(s)) due to the poor filter information of q_1's join units in data graph.

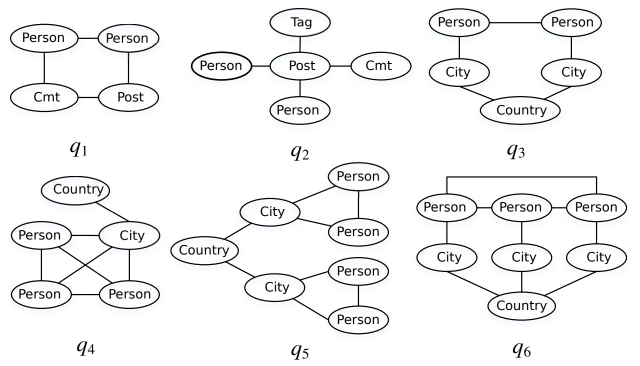


Fig. 7: Labelled Queries.

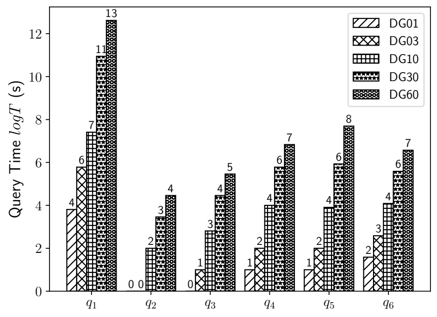


Fig. 8: All Labelled Queries.

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 shows that CliqueJoin++ has great scalability for labelled matching.

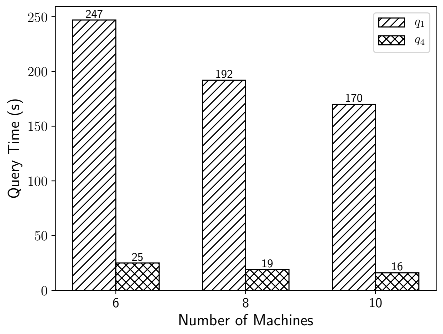


Fig. 9: Labelled Scalability.

V. Realted Work

Subgraph Enumeration. The first practical algorithm for subgraph enumeration was proposed by Ullmann in [38]. It is a backtracking algorithm which finds solutions by increasing partial solutions or abandoning them when it determines they can not reach to final results. Most of existing algorithms for subgraph enumeration follow Ullmann's backtracking approach. They exploit different join orders, pruning rules, and auxiliary information to narrow the search space, thereby enhance the performance. In particular, VF2 [14] and QuickSI [34] use *connected matching order* that generates the matching order by selecting a vertex connected to one of the already selected vertices rather than a random selection to prune false positive candidates as early as possible. GraphQL [21] and SPath [41] focus on reducing the candidates of query vertices by exploiting neighborhood-based filtering. TurboISO [19] and the boost technique in [31] propose to merge vertices with same labels and the same neighbours in q and G respectively to reduce the matching complexity. [27] 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, [32] studies 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 [39], Tree+Δ [42], and FG-Index [11]) and *exhaustive enumeration based approaches* (e.g., gCode [43], CT-Index [24], GraphGrep [16], GraphGrepSX [8], Closure-tree [20], and Grapes [15]). In approximate graph containment search, TALE [37] 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 CliqueJoin++ 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 MapReduce 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 CliqueJoin++ is up to 10 times faster than CliqueJoin for unlabelled matching, and has excellent performance and scalability doing labelled matching.

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] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

[37] 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.

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

[39] 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.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

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

[42] 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.

[43] 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.