# Optimized Distributed Subgraph Matching Algorithm based on Partition Replication


## ABSTRACT

Subgraph matching can help users to extract valuable information from huge amounts of data. With the explosive growth of data scale in the era of large data, the problem of subgraph matching for massive graph data has brought more challenges. At present, most of the subgraph matching algorithms utilize the recursive backtracking to match the query candidates gradually, and filter the query candidates with efficient index structures. However, most of such algorithms lacks of extensibility and scalability when dealing with tens of millions of graph data. With the development of distributed computing technology, the large-scale graph data can be partitioned in a distributed cluster, which can use the storage capacity and computing power of distributed technology to resolve the subgraph matching problem in the massive graph data.

At present, distributed subgraph matching is usually processed by RDF graph engine and Map-Reduce computing framework, which is difficult to satisfy the efficiency. We propose a distributed subgraph matching algorithm based on Partition Replica (noted as PR-Match) to process the partition and storage of large scale data graph. In the PR-Match algorithm, partition communication is pruned by the strategy of vertex neighbor replication, the query candidate set is reduced by means of vertex code, and the heuristic rule based on the prediction overhead is used to determine the merging sequence of sub-query matching results. Abundant of experiments on real data sets and synthetic datasets demonstrate the high efficiency and strong scalability of PR-Match algorithm when handling large scale data graphs.



## CCS CONCEPTS

• **Information systems** → *Information storage systems*; *Information storage technologies*;

## KEYWORDS

Subgraph Matching, Graph Indexing, Distributed Computing, Graph Partition

**ACM Reference Format:**

. 2019. Optimized Distributed Subgraph Matching Algorithm based on Partition Replication. In *Proceedings of The 12th ACM International Conference on Web Search and Data Minine (WSDM'19).* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn



Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).



*WSDM'19, February 11-15 2019, Melbourne, Australia*



© 2019 Copyright held by the owner/author(s).



ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn


## 1 INTRODUCTION

Graph is a semi-structured data represented by vertices and edges, which is usually represented as $G(V, E)$, where $V$ represents the set of vertices and $E$ the set of edges between vertices. In data analysis area, vertices usually used to represent objects while edges reflecting relationships among objects. For example, the protein can be viewed as the vertex of a graph and the interaction of protein can be regarded as the edge of a graph, a protein interaction network can be structured as STRING.

The graph theories has been applied to many scenarios such as optimal transport path, semantic web analysis, social network analysis, community discovery, knowledge question and answer, and so on. Subgraph matching is a very important research topic in the graph theory which can help the users to extract valuable information from the graph data sets. For example, in the biological protein networks, the protein molecular structure can be find to match the virus antibody. In the knowledge map, we can complete the knowledge retrieval with the subgraph matching by transforming the user description into the corresponding subgraph matching template.

Subgraph matching has been proved as a NP complete problem by Ullmann[20]. With the explosive growth of data scale in the big data era, the problem of subgraph matching for the massive graph data has brought more challenges. At present, most of the subgraph matching algorithms utilizes recursive backtracking method to match the query vertex continuously, and filter the query candidates with the graph index. However, most of the algorithms lacks of extensibility and scalability for the large scale graph data. Accompanying with the increment of graph size, single machine has difficulty in holding and computing large graph, therefore, the distributed graph storage and matching present the significance and necessity for massive graph data sets. Large graph data partition is a very important research direction in the distributed graph data processing. Existing distributed subgraph matching mainly uses RDF graph engine and map-reduce computing framework, which can hardly achieve a satisfied efficiency.

Since graph data is highly coupled and interactive, and the graph data processing is usually accessed according to the topological structure of the graph data, an optimized graph partitioning algorithm should not only ensure that the data size of the partitioned subgraphs is roughly the same, but also minimize the number of interactive edges between subpartitioned subgraphs. To solve these problems, we propose a distributed subgraph matching algorithm based on Partition Replica (noted as PR-Match) to process the partition and storage of large scale data graph. For the proposed PR-Match algorithm, we design a large-scale data graph partition and storage scheme based on the theory of equilibrium separation of large graphs, develop an high efficient vertex code index to process fast updating and maintenance on dynamic graphs, and establish

the heuristic rules based on the prediction overhead to determine the merging sequence of sub-query matching results.

We conduct abundant of evaluation and comparative experiments on the proposed PR-Match algorithm. The experiment results and analysis demonstrate that the PR-Match algorithm has a good scalability with difference size of graph data set. In addition, the performance of the PR-Match algorithm is greatly improved when the vertex average degree of the data graph is large and the lables are more signed, which shows the feasibility and efficiency of the vertex code index. At the same time, the query response time of the PR-Match algorithm can be only slowly increasing with the size of graph data set, and has obvious advantages over high performance graph database Neo4j, which can show that the PR-Match algorithm can be competitive for the large scale graph data matching.

The remaining part of this paper is organized as follows. Section 2 discusses the related work. Section 3 illusrates problem definitions about the subgraph matching. The proposed PR-Match algorithm is elaborated in Section 4. The experiments are elaborated in Section 5. Section 6 concludes the paper.

## 2 RELATED WORK

Subgraph matching usually use the index strategy which establishes the inverted index according to some features in the graph data to reduce the search space. iGraph[7] has made a related summary and introduction to the index based subgraph matching algorithm. iGraph divides the graph index set into mining index[3, 18, 21] part and non-mining index[22, 23] part. Mining index uses the high frequency subgraph mining algorithm to find high frequency subgraph, subtree, path and so on as the key of the index. gIndex[21] and FG-Index[3] generate the subgraph with edge number from 1 to maxL gradually. gIndex dynamically adjust the support degree for subgraphs with high frequency and subgraph identification containing relationships, therefore the size of the index space can be controlled. Different from the gIndex which only partially selecting the high frequency subgraph with strong filtering capability as the index structure, FG-Index uses all the high frequency subgraphs as index objects. Besides, in order to handle the oversize index space problem, FG-Index implements memory index layer and disk index layer so as to reduce the I/O overhead in the matching process. The Tree+Delta and SwiftIndex[22] take subtree as index structure. Tree+Delta uses the ring structure in the graph query as the online index and obtains a good retrieval effect for the graph query containing the ring. Non-mining index uses the graph inherent structure information to establish the index structure. C-Tree[22] puts forward the concept of the closure graph, and uses this concept to build a hierarchical tree. gCode[23] calculates the digital signature of each vertex in graph G for each data graph G, then, the digital signature of the graph G is generated according to the vertex signature, so that the GCode-Tree is constructed. iGraph has concluded that (1) although gIndex is the oldest index method, gInex has the best index effect (the pruning ability and I/O overhead in query); (2) the index effect is better in the dense data graph with fewer lables; (3) the results of C-Tree is a little poor in most cases; (4) the complex query Tree+delta on dense data graph is the best.

Previous algorithms based on the recursive backtracking and graph index target at small scale graph data sets. While billion scale social network and bioinformatics network is common at present, a lot of researchers begin to study the matching problem of super large scale graphs. Turboiso[6] algorithm proposes a plan merging candidate region detection and joint arrangement that makes the algorithm applicable to various query graphs with different structures and distribution. Similar paper[1] avoids the Descartes Cartesian product by means of using paths. Liang et al.[9] design a very clever index structure, which can take advantage of the powerful anti-monotone pruning, horizontal pruning and vertical pruning of the index structure to greatly reduce the candidate set. On this basis, a subgraph matching algorithm SMS2 is developed which can handle subgraph queries over tens of millions of vertices. Splitting a query graph into multiple sub query graphs accelerates the query procedure. Based on that, SGMatch[16] realizes high performance large scale subgraph matching by optimizing the query graph decomposition and prediction based subquery sequence.

With the development of cloud computing and distributed computing, more and more large-scale data processing and analysis are moving to the distributed environment which makes the research on distribution subgraph matching become hot. There are two main patterns of graph expression, simple graph and RDF graph, which can be transformed into each other although each of them owns different data presentation. Distributed RDF graph matching methods is divided into three categories according to[15] : cloud based method[11, 14] , partition based method[4, 10, 13] and joint based method[17]. Cloud computing methods mostly use map-reduce computing framework and HDFS-like distributed storage system. Partition based methods divide a RDF graph into multiple subgraphs and each subgraph is maintained by a cluster node. When a SPARQL query is proposed, it is splited into multiple subqueries and then merged the sub results to obtain the complete matching result. GraphPartition[13] splits the data graph by hash partition algorithm. The subgraph is extended by n-hop so that each subquery does not need to communicate with other cluster nodes. Joint based method need to get metadata for each RDF graph endpoint, which is suitable for data sharing among multiple organizations.

## 3 PROBLEM DEFINITION

We consider the subgraph matching problem on the labeled undirected graphs, and the relevant problem definitions are presented as follows:

*Definition 3.1.* **Label graph** A labeled graph is a quadruple of the form $G(V, E, L, F)$, where $V$ is a set of vertices, $e(u_i, u_j) \in E$ is a set of edges, $L$ is a set of label on vertices and edges, and $F$ is a labeling function of the form $F : V \cup E \rightarrow L$, such that it give a label to each vertex and edge.

*Definition 3.2.* **Subgraph** Given graphs $G_1 < V_1, E_1, L_1, F_1 >$ and $G_2 < V_2, E_2, L_2, F_2 >$, graph $G_1$ is the subgraph of graph $G_2$ if and only if:

(1) $V_1 \subseteq V_2, E_1 \subseteq E_2, L_1 \subseteq L_2$;
(2) $\forall v \in V_1, F_1(v) = F_2(v)$;
(3) $\forall e(v_1, v_2) \in E_1, F_1(e) = F_2(e)$.

Subgraph matching aims to find the graph which is isomorphism of the query graph in the graph data sets. The graph isomorphism definition is defined as follows:

*Definition 3.3.* **Graph isomorphism** Given graphs $G_1 < V_1, E_1, L_1, F_1 >$ and $G_2 < V_2, E_2, L_2, F_2 >$, $G_1$ and $G_2$ is graph isomorphism if and only if there is an injective function $f : V_1 \rightarrow V_2$ such that the conditions hold:

(1) $\forall v \in V_1, F_1(v) = F_2(f(v))$;

(2) $\forall e_1(v_1, v_2) \in E_1, \exists e_2(f(v_1), f(v_2)) \in E_2, F_1(e_1) = F_2(e_2)$.

For convenience, we use $Q$ to represent the query graph, $D$ to represent the graph database, $G$ to represent a data graph, $u$ to represent the vertex in the query graph, and $v$ to represent the vertex in the graph data.

## 4 PROPOSED PR-MATCH ALGORITHM

This section gives a detailed illustration about our proposed PR-Match algorithm, mainly invloving data graph partition and storage, query graph split, subquery match, and intermediate results merge.

### 4.1 Graph data partition

For the graph data partitions which are stored on multiple machines, the time cost of each subgraph matching is different. Considering the storage space overhead and the characteristics of the data access, we choose the neighbor vertex replication strategy to reduce the cost of subgraph matching. The graph data is firstly divided into the hash partition graph according to the specific vertex information. Then, the neighbor vertices of the core vertices in the partitioned graph and the direct adjacency edges are copied to the current partition graph. A distributed graph definition is given below:

*Definition 4.1.* **Distributed graph** A distributed graph $G < V, E, L, F >$ consists of a set of partitions $F = \{F_1, F_2, \cdots, F_k\}$, where each $F_i$ is specified by $< V_c^i \cup V_e^i, E_c^i \cup E_e^i, L_i, F_i >$ $(i \in 1, 2, \cdots, k)$ such that :

(1) $V_c^1, V_c^2, \cdots, V_c^k$ is a partition of $V$, $\forall i, j \in 1, 2, \cdots, k, i \neq j, V_c^i \cap V_c^j = \emptyset$ , and $U_{i \in 1,2,\cdots,k} V_c^i = V$, $V_c^i$ is called as core vertex of $F_i$;

(2) $e(v_1, v_2) \in E_c^i$, where $v_1 \in V_c^i, v_2 \in V_c^i$, $E_c^i$ is called as core edge of $F_i$;

(3) $E_e^i$ is a set of crossing edges between $F_i$ and other partitions, $E_e^i$ is called as extended edge of $F_i$;

(4) $e(v_1, v_2) \in E_e^i$, where $v_1 \in V_c^i, v_2 \in V_c^j$ and $i \neq j$, $V_e^i$ is called as extended vertex of $F_i$.

The data graph is divided into multiple partitions after being hashed. Each machine stores a partition graph. A specific description of the hash partition and vertex neighbor replication on graph data is described in **Algorithm1**. Firstly, Algorithm1 determines which partition the current vertex $v_i$ belong to according to the vertex hash value, at the same time, the vertex $v_i$ is the core vertex of the partition $F_i$. Secondly, Algorithm1 obtains the edge set of the partition. If the vertices of the current edge $e_i(v_m, v_n)$ are both in the same partition $F_i$, add the edge $e_i$ to the partition $F_i$ as the core edge of this partition. If the vertices $v_m$ and $v_n$ of the current edge belong to different partitions $F_i$ and $F_j$, then the edge is added to the partition $F_i$ and $F_j$ as their extended edges. Furthur more, Algorithm1 adds $v_m$ to partition $F_j$ as extended vertex of partition $F_j$, adds $v_n$ to partition $F_i$ as extended vertex of partition $F_i$. The time complexity of Algorithm1 is $O(|V| + |E|)$, where $|V|$ is the number of vertex in the graph data, and $|E|$ is the number of edges in the graph data.

**Algorithm 1:** Graph Data Partition Algorithm

**input :** a data graph $G < V, E >$

**output :** a set of Partitions $F = \{F_1, F_2, \cdots, F_k\}$

```
for v_i ∈ V do
    p = hash(v_i)modk + 1;
    V_c^p = V_c^p ∪ {v_i};
for e_i(v_m, v_n) ∈ E do
    if hash(v_m) = hash(vn) then
        p = hash(v_m)modk + 1;
        E_c^p = E_c^p ∪ {e_i(v_m, v_n)};
    else
        x = hash(v_m)modk + 1;
        y = hash(v_n)modk + 1;
        E_e^x = E_e^x ∪ {e_i(v_m, v_n)} , E_e^y = E_e^y ∪ {e_i(v_m, v_n)};
        V_e^x = V_e^x ∪ {v_n} , V_e^y = V_e^y ∪ {v_m};
```

### 4.2 Query decomposition

After partitioning the big graph data into the cluster environment, a subdgraph on a cluster node is incomplete. If the query request is sent to a cluster node directly, the information of other subgraph should be obtained from other cluster nodes, which would increase the communication overhead among the cluster nodes. According to the neighbor replication strategy, each cluster node has all the direct neighbor information of its core nodes. Based on this knowledge, the query graph can be decomposed into several subquery graph, so that each subquery can be independently calculated on each cluster node to reduce the communication overhead. Firstly, we give the relevant definitions.

*Definition 4.2.* **Hopping number** Given a graph $G < V, E, L, F >$, the hop number between vertex $v_i$ and vertex $v_j$ is denoted as $hop(v_i, v_j)$, which is the minimum distance between $v_i$ and $v_j$ in the graph. Similarly the hop number between vertex $v$ and edge $e$ is denoted as $hop(v, e(v_i, v_j))$, which is "$min(hop(v, v_i), hop(v, v_j))$+1" meaning the minimal number of crossing edges that $v_i$ reaching to $e$.

We specify $hop(v_i, v_j) = 0$ when $v_i = v_j$, otherwise, when $v_i$ and $v_j$ are not reachable, $hop(v_i, v_j) = \infty$.

*Definition 4.3.* **Star graph** Graph $G < V, E, L, F >$ is called as a star graph, if and only if:

(1) $\exists v_0 \in V, \forall v \in V$ when $v \neq v_0, hop(v_0, v) = 1$, $v_0$ is called as the center point of graph $G$;

(2) $\forall e \in E$, $hop(v_0, e) = 1$ where $v_0$ is the center point of the graph $G$.

We use $G^*$ to represent the star graph. From the Definition 4.3, we can find that the star graph $G^*$ is a graph composed of a center point and all its direct neighbor vertices, and the edges of a center point to its neighbor vertices. To be convenient, the star graph is represented as $G^* < v_0 \cup N(v_0), E, L, F >$, where $v_0$ is the center point of $G^*$, and $N(v_0)$ is the neighbor vertex of $v_0$ .

Theorem 4.4. *A data graph G is partitioned into $F = F_1, F_2, \cdots, F_k$, if the query graph Q is a star graph, query graph Q can be answered independently on all partitioned graph $F_i$.*

Proof. When the query graph $Q$ is matched with the partition graph $F_i$, the starting query vertex is bound to the center point of the partition graph $F_i$, because all the neighbors and direct adjacency edge information of the center point exist in the partition graph $F_i$, so all the match results of the query graph $Q$ with the partition graph $F_i$ can be obtained. □

To avoid the communication overhead between cluster nodes during the process of subgraph matching, the original query graph is split into several star subgraphs according to the theorem 4.4. Although it is a NP-hard problem to split a query graph into multiple star graphs, there still remains a variety of resolution schemes to be obtained. To choose the best decomposition solution, some conditions should be take into consideration. One is that the less number of subqueries, the less calculation cost of subgraph matching. Another condition is that the fewer candidate results for each subquery, the less cost of intermediate result merging. According to these conditions, we define the center point selection function for a star query graph, represented as Selectivity:

$$Selectivity(u_1) = \frac{degree(u_1)}{freq(u_1.label) * \min\limits_{e(u_1,u_2)\in E} freq(u_2.label)}$$

Where the $degree(u)$ represents the degree of vertex $u$, and $freq(u.label)$ indicates the number of labels of the vertex $u$ appearing in the graph data. We choose a vertex with bigger degree and fewer candidate sets as the center point of the query graph. Based on the selection function, we propose a query graph decomposition algorithm, as shown in **Algorithm2**.

**Algorithm 2:** Query Graph Decomposition Algorithm

**input :** a query graph $Q < V, E, L, F >$
**output :** a set of star graph $T$

```
while E ≠ ∅ do
    S_max ← 0, u ← null;
    for u_i ∈ V do
        S_tmp = Selectivity(u_i);
        if S_tmp ≥ S_max then
            S_max = S_tmp;
            u = u_i;
    T = T ∪ {G*(u)};
    E = E/{e|e ∈ G*(u)};
    V = u/{v|v = u, ordeg(v) = 0};
```

In Algorithm2, the input is the original query graph, and the output is the decompostion result of the query graph. Because subquery decompostion is an edge coverage problem, our algorithm utilizes the edge traversal method to generate subquery. The algorithm firstly finds the highest selective vertex, then constructs the star query graph with the current vertex as the center point of the current subquery. $G^*(u)$ is a star graph with vertex $u$ as its center point. Next, the algorithm deletes the center point of the current subquery and the involved edges, as well as the vertex with 0 indegree. This algorithm finishes when all the edges of the query graph are covered by the subquery. The time complexity of Algorithm2 is $O(|V| \cdot \overline{deg(u)})$, where $|V|$ represents the number of the vertex in the query graph, and the $\overline{deg(u)}$) represents the average degree of the vertex in the query graph.

## 4.3 Subquery matching

After the original query graph is decomposed into sevaral star query graphs, the subquery matching requests are distributed to all the nodes in the cluster, therefore, the cluster nodes can complete the subquery matching according to the local data. Since the subgraph matching is a NP-complete problem, most algorithms use the "filter-refining" framework to accelerate the response time of subgraph matching. Firstly, the candidates which cannot satify the conditions are removed by the pre-designed filtering strategy. Secondly, the subgraph isomorphism test is appled to the remaining candidate sets. With the thought of the graph index, we design a vertex code to reduce the search space of the subquery.

The vertex candidate set of the query vertex $u$ named $C(u)$ in graph database $D$ consists of all vertices which contains $F_v(u)$ labels in the graph database. If the vertex $v$ of the data graph matches the vertex $u$ of the query graph, $|N(v)|$ is bigger than $|N(u)|$, where $|N(u)|$ is the number of neighbors of the vertex $u$.

*Definition 4.5.* **Neighbor label signature** The neighbor label signature of vertex $v$ is denoted by $Sig(v)$, which is represented by a tuple $< P_n(v), P_e(v) >$, where $P_n(v)$ is a label of multiple sets of all its neighbor vertex, $P_e$(v) is a label of multiple sets of edges between vertex and its neighbors, that is:
(1) $I \in P_n(v) \Rightarrow \exists v^{'} \in N(u), I = L_v(v)$;
(2) $I \in P_e(v) \Rightarrow \exists v^{'} \in N(v), e(v, v^{'}) \in E, I = L_e(e)$.

Theorem 4.6. *Given graphs Q and G. Under the bijective function f, Q is isomorphic to G. For any vertex u in graph Q, the neighbor label signature of vertex u is signed to be $Sig(u) =< P_n(u), P_e(u) >$. If $v = f(u)$ and its neighbor label signature is signed to be $Sig(v) =< P_n(v), P_e(v) >$, then they should satisfy:*

*(1) $P_n(u) \subseteq P_n(v)$;*
*(2) $P_e(u) \subseteq P_e(v)$.*

*then, the label of vertex v covers that of the vertex u.*

The theorem 4.6 is clearly stated that, vertex neighbor label signature contains the label information of the vertices around the vertex and their rough structure information, thus the candidate nodes can be filtered with the vertex label signature. In order to update and verify the signature information of vertex label, we map the signature information of the vertex neighbor label to the numerical space.

*Definition 4.7.* **Label code** Given a label $l$, the number of non-negative hash functions $m$, the label code of label $l$ is denoted by $Encode(l)$ which is a binary string $I$ with a length of $K$, where $I$ is initialized to 0, and each of the values satisfies the following formula: where $I[j]$ represents the value of the $j_{th}$ bit in the binary string $I$.

$$I[hash_i(label)] mod K + 1 = 1, i \in 1, 2, \cdots, k;$$

*Definition 4.8.* **Vertex code** Given a vertex $v$, the neighbor label signature of point $v$ is signed to $Sig(v) =< P_n(v), P_e(v) >$, and the vertex code of vertex $v$ is denoted by $Encode(v) = p \diamond q$, where $p$ is a counting string of all labels encoded in $P_n(v)$, and $q$ is a counting string of all labels encoded in $P_e(v)$. $\diamond$ is a join operation for counting strings, and $|Encode(v)| = 2k$, that is:

$$(1)\ p[i] = \sum_{l \in P_n(v)} Encode(l)[i], i \in 1, 2, \cdots, k$$
$$(2)\ q[i] = \sum_{l \in P_e(v)} Encode(l)[i], i \in 1, 2, \cdots, k$$

Theorem 4.9. *Given graphs $Q$ and $G$, under the bijective function $f$, $Q$ is isomorphic to $G$. For any vertex $u$ in graph $Q$, the vertex code of vertex $u$ is signed to be $Encode(u) = p_1 \diamond q_1$, if $v = f(u)$ and its vertex code is signed to be $Encode(v) = p_2 \diamond q_2$, then they should satisfy:*

$$(1)\ p_1[i] \leq p_2[i];$$
$$(2)\ q_1[i] \leq q_2[i].$$

According to the aboved definitions and theorems, the subquery matching algorithm is presented in **Alogirhtm3**. The star graph matching mainly includes two processes: namely off-line operation and online operation. In off-line operation, vertex code is generated for each vertex in the graph database $D$. The online operation is divided into two stages: candidates filtering and subgraph connectivity testing. The algorithm3 firstly obtains the candidate set of center point of star query graph based on the label, then removes the vertex candidates which are not the center point of the partition $F_i$, then the pruning operation is carried out according to the vertex degree and the vertex code, then finishes the connectivity test of the star graph. Finally, the Cartesian product of the neighbor matching vertices of the center points is computed and all the matching results are expanded. The worst time complexity of Algorithm3 is $O(n \cdot (m-1)!)$, where $n$ represents the number of candidate sets of the center point of the star query graph in the graph database $D_i$, and $m$ represents the number of vertex contained in the query graph.

**Algorithm 3:** Subquery Matching

**input :** star query graph $Q^* < u_0 \cup N(u_0), E_q, L_q, F_q >$, partition graph $F_i$ on clusters $i$
**output :** a set of matching graph $PM_i^q$ graph

```
PM_i^q ← ∅ ;
get candidate vertices C(u_0) of query vertex u_0 ;
for v_0 ∈ C(u_0) do
    if |N(v_0)| ≤ |N(u_0)| then
        Break ;
    if v_0 ∈ V_c^i then
        Get vertex code of vertex v_0 as Encode(v_0) ;
        if
         Encode(v_0)[i] ≥ Encode(u_0)[i], i ∈ {1, 2, 3, ··· , 2k}
         then
            for u_m ∈ N(u_0) do
                for v_n ∈ N(v_0) do
                    if F_q(u_m) ⊆ F_i(v_n) and F_q(u_0, u_m) ⊆
                     F_i(v_0, v_n) then
                        S_m → S_m ∪ {(u_m, v_n)};
            PM_i^q = PM_i^q ∪ {{(u_0, v_0)} × S_1 × S_2 × S_3 ··· S_p}
             ;
```

## 4.4 Intermediate result merge

After completing the previous work, the matching results of the subqueries of the original query graph are obtained. In order to achieve the result of the original query graph, it is necessary to merge the intermediate results of the subquery. The merge operation of the subquery matching results is a time-consuming task and a large number of previous work[6, 8, 12, 19] shows that the matching order of processing units and the merging order of subquery matching results have a very significant impact on the performance of subgraph matching. This section mainly discusses the optimization of the merging order of sub query matching results.

*Definition 4.10.* **Merge plan** The partition result of the query graph $Q$ is $T = q_1, q_2, \cdots, q_n$, and its matching result on all cluster nodes is $M = M_1, M_2, \cdots, M_n$, and $\Omega = M_{s1} \bowtie M_{s2} \bowtie \cdots \bowtie M_{sn}$ represents a merge plan for the matching result of subquery. The star graph corresponding to the $M_{si}$ has intersecting vertices with a subquery graph before $M_{si}$ in the merge plan sequences. $M_{si} \in M$, $\bowtie$ represents the merge operation.

In the definition 4.10, the subquery located in the merge plan is intersected with a certain subquery before its location, which can make sure that each merge performs a connectivity check and avoids the invalid merge overhead.

*Definition 4.11.* **Merge cost** A graph database $D$ has been stored on $m$ machines, the query graph $Q$ is decomposed into $n$ star query graph, the matching results of subquery $q_i$ on the node $k$ is $PM_k^i$, then the merge overhead of the merge plan $\Omega$ is:

$$Cost(\Omega) = O(\prod_{i=1}^{n}(\sum_{j=1}^{m}(|PM_j^i| + 1)))$$

*Definition 4.12.* **Optimal merge plan** Given the matching results of all subqueries on the partition, the merge plan is the optimal merge plan if and only if for any merge plan $\Omega^{'}$, $Cost(\Omega) \leq Cost(\Omega^{'})$.

Since finding the optimal merge plan is a NP-complete problem, many researcheres use dynamic programming and greedy strategy to obtain the suboptimal merge plan. The method to obtain the suboptimal merge plan are mainly divided into two categories. The first one is to determine the merge sequence before the actual merge conduction accroding to a static overhead prediction model, so the merge sequence will not be modified during the merge process. Actually, a well performed static overhead prediction is a key point in this method. The other one is to firstly choose an initial matching set, and the next matching set is dynamically selected according to the current merged state. This method requires a dynamic merging cost calculation model. Although the dynamic methods have better merging performance, the static methods have better results for specific data sets or specific query graphs. In this paper, we use the static method to determine the merging order of the subquery matching results, and design a static cost prediction function called as $P - Cost$.

*Definition 4.13.* **Prediction merge cost** A graph database $D$ has been stored on $m$ machines, the partition result of the query graph $Q$ is $T = q_1, q_2, \cdots, q_n$, and its matching result on all cluster nodes is $M = M_1, M_2, \cdots, M_n$, and $\Omega = M_{s1} \bowtie M_{s2} \bowtie \cdots \bowtie M_{sn}$ represents a merge plan for the matching result of subquery,, then the prediction merge cost of the merge plan $\Omega$ such that:

(1) The prediction merge cost of the matching result $M_{si}$ and the matching result $M_{sj}$ is:

$$P - Cost(M_{si} \bowtie M_{sj}) = (\sum_{i=1}^{m}(|PM_i^{si}| + 1)) \times (\sum_{i=1}^{m}(|PM_i^{sj}| + 1))$$

(2) The prediction merge cost of merging operation $O_i$ and matching result $M_{si+1}$ is:

$$P - Cost(O_i \bowtie M_{si+1}) = p - Cost(O_i \times (\sum_{i=1}^{m}(|PM_i^{si}| + 1)) \times (\frac{1}{2})^{\alpha})$$

(3) The prediction merge cost of merge plan $\Omega$ is:

$$P - Cost(\Omega) = \sum_{i \in 1,2,\cdots,n-1}(P - Cost(O_i \bowtie M_{i+1}))$$

$\alpha$ is the number of intersecting vertices of the subquery $q_{si+1}$ and the merging result $O_i$, represents the matching results of subquery $q_i$ on the node $k$, and the $M_{si} \in M$, $\bowtie$ represents the merge operation.

Since the size of the query graph is small, so we can list all possible merging plans in a reasonable time, then we can choose the merging plan with the lowest prediction cost as the optimal merger plan.

After the merging sequence is determined, we can use the nested loop to complete merging of the matching results of the subquery. The illustration about the intermediate result merge is presented in **Algorithm4**. The algorithm4 mainly completes the merge process of the matching results of the subquery by calling the merge subroutine *recusiveJoin* which is presented in **Algorithm5** based on the depth first traversal. The Algorithm5 firstly checks whether the current depth has reached the maximum depth (that is, the merge result is the matching graph of the original query graph). If the maximum depth is reached, the current result is added to the final result set, otherwise a connectable test will be conducted between the current depth subquery matching results and the merged results. The merge state will be updated when one of the current subgraph matching result is connectable with the merged result, then proceed to the next layer's recursive merge operation. Note that after each merge is completed, the state that needs to be restored before the merge would be returned. The worst time complexity of the matching result merge is $O(\prod_{i=1}^{m}(|M_{s_i}|)$, where$|M_{s_i}|$ represents the number of matching graphs of the subquery $q_{s_i}$ on all partition graphs.

**Algorithm 4:** Subquery Matching Result Merge

**input :** optimal merge plan $\Omega = M_{s_1} \bowtie M_{s_2 \bowtie \cdots \bowtie M_{s_n}}$, the matching results of all subqueries on all partitions $PM = \{\cup PM_i^{s_1}, \cup PM_i^{s_2}, \cdots, \cup PM_i^{s_n}\}$

**output :** all the matching subgraph $MG$ of original query graph on graph database $D$

1 $MG \leftarrow \emptyset, M \leftarrow \emptyset$, curDepth $\leftarrow 1$, maxDepth $\leftarrow n + 1$;
2 Call `recusiveJoin` (curDepth, maxDepth, M, $MG$) ;
3 return $MG$;

**Algorithm 5:** Merge Subroutine recursiveJoin

**input :** curDepth, current recursive depth; maxDepth, the max recursive depth, $M$, intermediate matching result; $MG$, matching subgraph

**output :**

1 **if** curDepth == maxDepth **then**
2  $MG \leftarrow MG \cup M$ ;
3 *Get the subquery matching results of* $q_{\text{curDepth}}$ *as* $\cup PM_i^{\text{curDepth}}$ ;
4 **for** $G_i^* \in \cup PM_i^{\text{curDepth}}$ **do**
5  **if** $G_i^*$ *is joinable with* $M$ **then**
6  Merge $G_i^*$ with intermediate result $M$, as $M \leftarrow M \bowtie G_i^*$ ;
7  `recursiveJoin` (curDepth + 1, maxDepth, $M$, $MG$) ;
8  Remove $G_i^*$ from $M$, and restore the state before merge ;

## 5 EXPERIMENTS

The proposed PR-Match algorithm is conducted in a distributed cluster with 6 machines, each of the machine is configured with 8G DDR3 memory, a Intel i5-4590 CPU of 3.3GHz, four core numbers per CPU, the network adapter is the RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller with a capacity of 1Gbps, a ST1000DM003-1ER162 disk with a capacity of 1TB. All the code of the algorithm is implemented by Java, the operating system of the machine is Ubuntu Linux, and the version number of the neo4j graph database used by the comparison experiment is neo4j-community-3.1.7. The experiments data sets are illustrated as below:

(1) The subgraph matching on the small graph set uses the AIDS real data and the synthesized dataset generated by GraphGen;

(2) The subgraph matching on a single large graph uses the US Patents[5] real dataset and the synthesized dataset generated by R-Mat[2] .

### 5.1 Subgraph matching on small graphs

For the subgraph matching experiment of small graph sets, we use the AIDS data set and the GraphGen synthesis data set. GraphGen synthetic data set contains 20,000 data graphs, 100 vertex labels and 100 edge labels. The experimental results are shown in Figure 1 and Figure 2. It is found that the efficiency of the subgraph matching of PR-Match algorithm is similar to that of the Neo4j. When the query graph is small, the PR-Match even performs worse than the Neo4j because Neo4j is a centralized single machine matching and the PR-Match algorithm is a distribution pattern matching.

### 5.2 Subgraph matching on a single large graph

US Patents is a patent reference network that recorded the reference relations of patents between 1963 and 1999 in the United States. It is used in[19] . We use the patent "NCLASS" domain as a patent label, then 714 labels of vertex in total. For the reason that there is only one reference relationship between the patents, to extend the edge relationship, considering a relationship edge, we use the sum of the end point patents release year as the edge label, so we

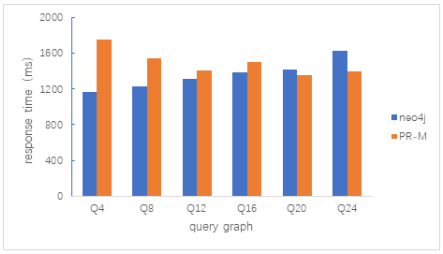


**Figure 1: Subgraph matching on a set of small graphs-AIDS**


**Figure 2: Subgraph matching on a set of small graphs-GraphGen**

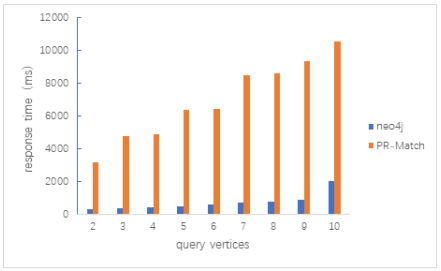


**Figure 3: Path matching-US Patents**

get 630 edge labels in total. R-Mat is a large graph generating tool to simulate large scale network graphs. The graph generated by R-Mat is unlabeled graph. For this reason, we randomly assign a label for each vertex and edge. Three types of query graphs for subgraph matching have been conducted on a single large graph: path query, clique query and random graph query.

*5.2.1 Path query.* In the path query experiment, the query graph is a path consists of vertices. We give a path set which contains 9 kinds of path, the vertex number of path range from 2 to 10. Experiment results on the US Patents data set and the R-Mat synthetic data set are shown in Figure 3 and 4. Accroding to the Figure 3 and Figure 4, neo4j has advantages in path matching while the response time of PR-Match algorithm is a little high in the path matching. The main reason is that neo4j uses a unique physical storage mode and a powerful traverse framework, meanwhile, PR-Match increases the number of sub-query in the path query and the pruning ability at the center vertex of the sub-query also decreases.

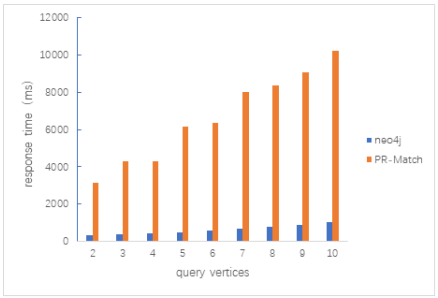


**Figure 4: Path matching-R-Mat**


**Figure 5: Clique matching-US Patents**


**Figure 6: Clique matching-R-Mat**

*5.2.2 Clique query.* In clique query, each query graph is a complete graph, which indicates that the query graph contains $1/2|V|(|V|-1)$ edges. The clique query graph set contains 6 kinds of queries with the graph vertex number vary from 2 to 7. Experiment results on the US Patents data set and the R-Mat synthetic dataset are shown in Figure 5 and 6. With the analysis, Neo4j query response time is rapidly increased when the vertex number of the query graph goes big. As a comparison, the PR-Match response time growth maintains as a stable rate. The query efficiency of neo4j is lower than PR-Match when the vertex number of a query graph is larger than 5. This is because as the query graph density increases, the number of query resolution will not be too large but the neighbor label density vertex increases at the same time. The pruning ability of vertex neighbor label coding is highly improved.

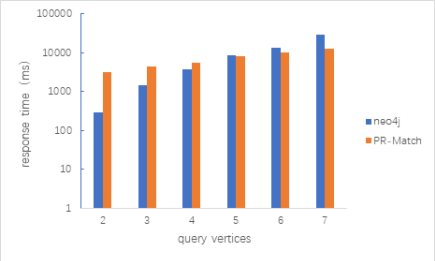


**Figure 7: Random query-US Patents**

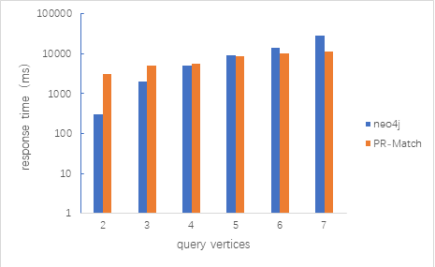


**Figure 8: Random query-R-Mat**

*5.2.3 Random query.* The vertices and edges of the query graph in the random graph query are randomly selected. The random query graph set contains a total of 6 kinds of queries with the number of vertices ranging from 2 to 7. Experiment results on the US Patents data set and the R-Mat composite data set are In Figure 7 and 8, the PR-Match algorithm does not have advantages over the neo4j in the small-scale query graph, but as the size of the query graph gradually increases, the gap between PR-Match and neo4j has been narrowed and surpassed.

## 5.3 Scalability test of PR-match algorithm

Two sets of experiments are designed to test the scalability of subgraph matching. The first set is used to study the effect of data graph size on the efficiency of subgraph matching. The second set aims to study the influence of average vertex degree of the query graph on the efficiency of subgraph matching.

*5.3.1 Data size.* To study the effect of different scale data graphs on the efficiency of subgraph matching, we use GraphGen to generate 5 sets of small graphs, the graph number of each set are 10K, 20K, 30K, 40K and 50K respectively. We also get 5 large graphs generated by R-Mat, the number of vertices corresponding to the 5 large graphs are 2 million, 4 million, 6 million, 8 million and 10 million respectively. The used query graph is a random graph. The query graph contains 7 vertices. The experimental results are presented in Figure 9 and 10. Comparative analysis shows that PR-Match has obvious advantages in large-scale data graphs and query response time increases slowly with the increase of data graph size.

*5.3.2 Average vertex degree.* In order to study the influence of the average vertex degree of the query graph on the subgraph matching algorithm, the experimental evaluation is carried out on the AIDS and US Patents data sets. The query graph is a random

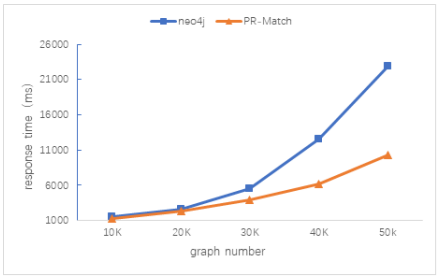


**Figure 9: Data size-GraphGen**

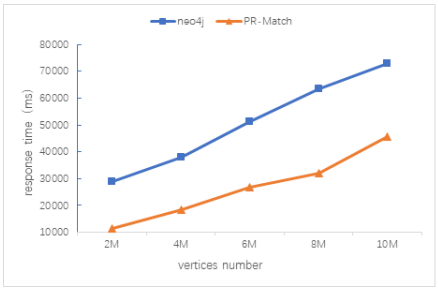


**Figure 10: Data size-R-Mat**

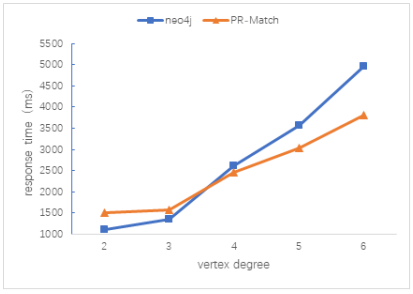


**Figure 11: Average vertex degree-AIDS**

graph, and the query graph contains 7 vertices with average vertex degrees increased from 2 to 6. Experiment results shows in Figure 11 and 12. Comparative analysis shows that on the small scale data graph AIDS, the performance of Neo4j and PR-Match is similar but the response time of the PR-Match algorithm increases slower than Neo4j when the average vertex number of query graph increased. On the large data graph US Patents, PR-Match is not only owns shorter response time but also has lower response time increament rate when the average vertex degree of the query graph grows compared with Neo4j. Therefore, we can demonstrate that PR-Match algorithm has obvious advantages in large scale data graph and dense query graph.

## 6 CONCLUSIONS

Our paper proposes a PR-Match algorithm for subgraph matching on large scale graph data sets.The main work involves that a vertex neighbor replication strategy is designed to consider the efficient graph data partition and query decomposition; a vertex code graph

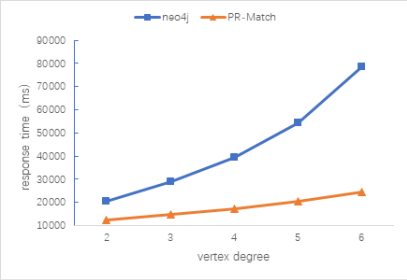


**Figure 12: Average vertex degree-US Patents**

index of the vertex neighbor label is used to prune the query result candidate set; a combined order selection strategy based on the cost prediction is proposed to greatly reduces the cost of merging. The abundant experiments are conducted to domonstrate the efficienty and scalebility of the proposed PR-Match algorithm.

## REFERENCES


[1] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *International Conference on Management of Data*. 1199–1214.

[2] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Siam International Conference on Data Mining, Lake Buena Vista, Florida, Usa, April*.

[3] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. 2007. Fg-index:towards verification-free query processing on graph databases. In *SIGMOD*. 857–872.

[4] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. *TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing*. 289–300 pages.

[5] Bronwyn H. Hall, Adam B. Jaffe, and Manuel Trajtenberg. 2001. The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools. *Nber Working Papers* (2001).

[6] Wook Shin Han, Jinsoo Lee, and Jeong Hoon Lee. 2013. Turbo iso : towards ultrafast and robust subgraph isomorphism search in large graph databases. In *ACM SIGMOD International Conference on Management of Data*. 337–348.

[7] Wook Shin Han, Jinsoo Lee, Minh Duc Pham, and Jeffrey Xu Yu. 2010. iGraph: a framework for comparisons of disk-based graph indexing techniques. *Proceedings of the Vldb Endowment* 3, 12 (2010), 449–459.

[8] Huahai He and Ambuj K. Singh. 2008. Query Language and Access Methods for Graph Databases. In *ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, Bc, Canada, June*. 405–418.

[9] Liang Hong, Lei Zou, Xiang Lian, and Philip S. Yu. 2015. Subgraph Matching with Set Similarity in a Large Graph Database. *IEEE Transactions on Knowledge & Data Engineering* 27, 9 (2015), 2507–2521.

[10] Katja Hose and Ralf Schenkel. 2013. WARP: Workload-aware replication and partitioning for RDF. In *IEEE International Conference on Data Engineering Workshops*. 1–6.

[11] Mohammad Husain, James Mcglothlin, Mohammad M Masud, Latifur Khan, and Bhavani M Thuraisingham. 2011. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Transactions on Knowledge & Data Engineering* 23, 9 (2011), 1312–1327.

[12] Jinsoo Lee, Wook Shin Han, Romans Kasperovics, and Jeong Hoon Lee. 2013. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the Vldb Endowment* 6, 2 (2013), 133–144.

[13] Kisung Lee and Ling Liu. 2013. Scaling queries over big RDF graphs with semantic hash partitioning. *Proceedings of the Vldb Endowment* 6, 14 (2013), 1894–1905.

[14] Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras, and Nectarios Koziris. 2014. H 2 RDF+: an efficient data management system for big RDF graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*.

[15] Peng Peng, Lei Zou, Lei Chen, and Dongyan Zhao. 2016. Processing SPARQL queries over distributed RDF graphs. *Vldb Journal — the International Journal on Very Large Data Bases* 25, 2 (2016), 243–268.

[16] Carlos R. Rivero and Hasan M. Jamil. 2016. *Efficient and scalable labeled subgraph matching using SGMatch*. Springer-Verlag New York, Inc. 1–27 pages.

[17] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. 2011. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *International Conference on the Semantic Web*. 601–616.

[18] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the Vldb Endowment* 1, 1 (2008), 364–375.

[19] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *Proceedings of the Vldb Endowment* 5, 9 (2012), 788–799.

[20] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *Journal of the Acm* 23, 1 (1976), 31–42.

[21] Xifeng Yan, Philip S. Yu, and Jiawei Han. 2004. Graph indexing:a frequent structure-based approach. 335–346.

[22] S. Zhang, M. Hu, and J. Yang. 2007. TreePi: A Novel Graph Indexing Method. In *IEEE International Conference on Data Engineering*. 966–975.

[23] Lei Zou, Lei Chen, Jeffrey Xu Yu, and Yansheng Lu. 2008. A novel spectral coding in a large graph database. In *EDBT 2008, International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*. 181–192.