Regular Expression Matching on billion-nodes Graphs

Hongzhi Wang, Member, IEEE, Jiabao Han, and Bin Shao, Jianzhong Li, Member, IEEE

Abstract—In many applications, it is necessary to retrieve pairs of vertices with the path between them satisfying certain constraints, since regular expression is a powerful tool to describe patterns of a sequence. To meet such requirements, in this paper, we define regular expression (RE) query on graphs to use regular expression to represent the constraints between vertices. To process RE queries on large graphs such as social networks, we propose the RE query processing method with the index size sublinear to the graph size. Considering that large graphs may be randomly distributed in multiple machines, the parallel RE processing algorithms are presented without the assumption of graph distribution. To achieve high efficiency for complex RE query processing, we develop cost-based query optimization strategies with only a small size statistical information which is suitable for querying large graphs. Comprehensive experimental results show that this approach works scale well for large graphs.

1 INTRODUCTION

Graph data have been widely applied in many areas such as knowledge management [20], social network [1], bioinformatics [3], and compilers [21].

In the area, an important application is to retrieve pairs of vertices with specific path between them. For instance, in knowledge management, the information of whether two persons with common characteristics come from the same lineage may be required to extract. It is hard to express by a subgraph or SPARQL query.

As a powerful tool of string pattern description, regular expression can also be used to describe the relationship of vertices in a graph. RE allows querying of arbitrary length paths by using regular expression patterns, it is useful for expressing complex navigation in a graph, in particular, union and transitive closure are crucial when one does not have a complete knowledge of the structure of the knowledge base. In our example, suppose a person M and a person N share a common characteristic. In other words, they share a common attribute values in the knowledge database. If we want to know whether M and N share to the same lineage. This query can be expressed by regular expression easily.

Even some existing work [6], [7], [9], [10], [12], [18], [19] start to study regular expression on graphs, their methods have two shortcomings.

One one hand, existing approaches only consider a share of operations in regular expressions. [6] only studied the regular expression matching on the graph without “or”(∪) and recursion closure. [9], [10], [12] considers only the regular expression matching on tree structure data. Thus, the range of application for these methods is limited.

On the other hand, the scalability is not considered sufficiently. Currently, a graph may scale to very large, even to billion-nodes. For example, FreeBase, the online query processing and services on frequently updating graphs with large size requires lightweight indices. However, [6] requires the structural index with size at least the number of edges of graph. Even though it works efficiently on graphs with small size, it is not suitable for large graphs. [7], [18], [19] do not consider the parallelization of regular expression matching, and they could hardly handle very large graphs that could not reside in a single machine.

To process flexible regular expression on billion-nodes graphs with only a lightweight index, in this paper, a search-based method is proposed. In this method, only an index that retrieves the vertices by their labels is used, which is in size linear to the number of nodes and easy to update. Without structural index, our method is traversal-based.

To avoid large intermediate results, in our system, a lightweight representation of intermediate results is designed. Based on such representation, we propose the basic operators for regular expression processing. For the scalability issues, besides implementation in a single computer, we propose parallel implementation of the operators in a cluster. To find an efficient execution order of the basic operators, a cost-based query optimization strategy for regular expression is proposed with a cost model which requires only small statistics information of graphs. The contributions of this paper are summarized as follows.

• The regular expression query processing on a large graph with full features is studied in this paper. To the best of our knowledge, this is the first paper that studies parallel processing of the full features of regular expression on large graph data.
• With the consideration of the requirement of large graph, we propose a traversal-based strategy for regular expression processing on a large graph. For the convenience of query optimization, we divide query processing into operators. For each operator, efficient implementation algorithms on both single computer and clusters are developed.
• For query optimization of regular expression queries on graph, a cost model is proposed. With the consideration of the feature of large graphs, an estimation strategy requiring only simple statistic information is proposed.
• A dynamic programming algorithm is designed to obtain efficient execution order of operators for regular expression processing based on the estimated cost.
• Extensive experimental results demonstrate that our approaches could accomplish regular expression matching on billion-nodes graphs within 2 seconds and are suitable for various graphs and queries.

Section 2 describes the query language. In Section 3, the framework of regular expression processing is proposed. The logical operator generation is described in Section 3.3. The implementation algorithms of the operators are presented in Section 4. The cost model and query optimization strategy are described in Section 5. Section 6 summaries related work and Section 8 concludes the whole paper.

2 Query Language

In many scenarios, the relationship between two vertices in a graph may be complex. In a knowledge base, the relationship between vertices could be naturally represented as the label sequence on the path between them. Thus, to retrieve vertex pairs satisfying requirements in a graph, the query language should describe the label sequence required to be between two vertices. However, with the increase of path length, the relationship between two vertices becomes too complex to be expressed with compact queries with simple semantics.

For instance, N is the offspring of M. Obviously, they share the same lineage. In the specific case, we can hardly obtain the length of the path between M and N. Thus, it is difficult to express this relationship.

Regular expression is a form of language for sequence in common use. Similar as the regular expression for strings, a regular expression can describe labels attached to the vertices in a path between two vertices in a labeled graph. Hence, a regular expression can be used to describe the requirement of the paths between two vertices in a graph. In the above example, each person has a property ‘status’, it has a label p if he or she has a child and everyone has a label c. If M and N come from the same family, the path between M and N must satisfy a criteria: \( p \rightarrow c \rightarrow p \rightarrow c \rightarrow \ldots \), where we use ‘→’ to denote concatenation to avoid confusion. If we use regular expression to express this criteria, it can be expressed like this: \( p(c \rightarrow p)^+ c \). Note that reachability queries cannot express such query properly since reachability queries neglect the label in the path.

Formally, the regular expression is defined as following. For the flexibility, the wildcard representing any label is added to the definition.

Suppose the tag set is \( \Sigma \). The wildcard is denoted by “#”. The regular expression on \( \Sigma \), denoted by \( RE(\Sigma) \), is defined as follows.

\[
\begin{align*}
\Sigma \cup \{\#\} & \subseteq RE(\Sigma) \\
\forall r_1, r_2 \in RE(\Sigma), r_1 \cdot r_2 & \in RE(\Sigma) \\
\forall r_1, r_2 \in RE(\Sigma), r_1 | r_2 & \in RE(\Sigma) \\
\forall r \in RE(\Sigma), r^+ & \in RE(\Sigma) \\
\text{No other expressions are in } RE(\Sigma)
\end{align*}
\]

A data graph is defined as a graph \( G=\langle V_G, E_G, T_V \rangle \), where \( V_G \) is the vertex set, \( E_G \) is the edge set, and \( T_V : V_G \rightarrow \Sigma \cup \# \) assigns a label to each vertex in \( V_G \).

If the string \( s_p \) constructed by connecting the tags in the vertices of a path \( P \) from the start node to the end node matches a regular expression \( E \), it is said that \( P \) matches \( E \).

The result of a regular expression \( E \) on a data graph \( G \) is a set of pairs \( R_{E,G}=\{(v_i,v_k)| P \text{ exists in } G \text{ matches } E\} \).

For example, consider the data graph shown in Figure 1. The results of the RE query \( Q_1=a(b|e|cd)^+f \) are \( \{(a_1, g_1), (a_2, g_2), (a_3, g_3)\} \). For \( a_1 \) and \( g_1 \), the path between them \( a_1b_1c_1g_1 \) match the path \( a_{g}b_{e}c_{d}g_{f} \) in the RE. The path \( a_1c_2d_3e_3f_3g_3 \) match the path \( a\cdots + f \) in the RE.

3 The Framework of RE Query Processing

In this section, we present the framework of RE query processing. Since our goal is to process RE query on large graphs, only lightweight indices with size linear or sublinear to the graph size are permitted. As discussed in [16], with only lightweight index, only simple operators are supported for structural query processing, e.g., traversal and join. RE query processing is to combine such simple operators. Thus, we attempt to process a RE query \( r \) by separating \( r \) into segments properly, each of which is processed by traversal separately, and joining the results of these segments.

With the consideration of the complexity of RE query, it is difficult to perform the query splitting directly. Thus, we handle the query in two tiers. The first one converts a query to the logical plan, which is near to the form of the query. The second one is to generate the physical plan from the logical plan. Then we discuss these two tiers, respectively.

3.1 Logical Operators

Since a RE includes three basic operations, concentration, alternative operator | and closure operation: +, it is natural to match regular expression by turning these basic operations to the matching in a graph. Thus, we define logical operators to support the matching according to the operations in RE.

In RE, the operations concentration and closure are used to describe the connection relationships between nodes in the graph. Concentration means the directly connection between two nodes in a path. Closure means the emergence of node sequences with same label sequence in a path consecutively and repeatedly. The logical operators for RE are defined according to these two operations. To support
the operation of ‘|’, each variable in the operator is permitted to match multiple labels in the RE.

The input of the operators may be the nodes matching some labels in the RE or the intermediate results matching a sub expression. With the consideration of input source, corresponding operation in RE and search direction, we summarize 6 logical operators as shown in Table 1, where each lowercase letter is a variable referring to a set of labels in RE. In this table, we use ‘→’ to represent ‘→’ or ‘←’, which identify the direction of the execution.

Intuitively, the processing of an RE query can be converted to a series of logical operators. The logical operators in a regular expression may have multiple possible execution orders. How to generate an efficient execution order is the task of query optimization, which will be discussed in Section 5.

Example 1 demonstrates the semantics of the logical operators and the framework of regular expression query processing.

Example 1. Consider the query \( a(b|e)(cd)^+ f \)g that is processed in from left to right. The first operator is \( a \rightarrow (b|e)\{b, c\} \). The results of this operator are the set of path fragments, each of which matches \( a - b \) or \( a - c \). Here \( b|e \) is a regular expression. \( \{b, c\} \) means that \( a \) could be connected to a \( b \) or \( c \) node.

The following operator is \( a(b|e)(\{b\}) \rightarrow e \). It means that the \( b \) node in a result of \( a(b|e) \) connects to an \( e \) node. In Figure 1 the results of the subquery \( a(b|e) \) are \( \{a_1, b_1\}\{a_1, c_1\}\{a_1, c_2\} \). The partial results of \( a(b|e)(\{b\}) \rightarrow e \) are \( \{a_1, e_1\}\{a_1, c_1\}\{a_1, c_2\} \), corresponding to the regular expression \( E_1 = a(b|e) \).

The following operator is closure \( (c \rightarrow d)^+ (E_1(c), d) \). The intermediate results in this step are \( \{(a_1, e_1), (a_1, d_1), (a_1, d_2), (a_1, d_3)\} \) as the results of partial query \( E_2 = a(b|e)(cd)^+ \) By processing the operator \( E_2(d) \rightarrow f \), the intermediate results \( \{(a_1, e_1), (a_1, f_1), (a_1, f_2), (a_1, f_3)\} \) are obtained for the partial query \( E_3 = a(b|e)(cd)^+ f \).

Then the last operator \( E_3\{\{c, f\}\} \rightarrow g \) is processed to retrieve the results for the query.

3.2 Physical Operators

To efficiently execute logical operators on large graphs, in this section, we summarize 6 physical operators for them. Each of these operators could be implemented with lightweight index or in absence of index.

Note that when one label exists multiple times in a RE, during query processing, the two existences are treated as two labels and the intermediate results are maintained independently to avoid the confusion. For example, for the RE \( b(c^+) \), the first label \( c \) and the last label \( c \) in the closure may refer to different node set during query processing. Hence they should be distinguished.

Intuitively, the basic physical operators load nodes from the graph according the label and loading the neighbours with some specific labels. Two operators, Load and Neighbour, are defined for them, respectively.

We also propose two physical operators to link the intermediate results. One links the nodes matching some labels to intermediate results (SingleLink). The other joins of two groups of intermediate results (DoubleLink).

![Fig. 2. Intermediate Results](image)

Corresponding to these two logical operators related closure, two special physical operators, ClosureLink and SelfLink, are required as the fixed point of finding the repeated path matching the closure.

According to above discussions, physical operators are summarized in Table 2, where each variable may correspond to a set of labels, as is similar as logical operators. For example, for an RE \( E=(a|bc|def) \), head\( (E) = \{a, b, d\} \), tail\( (E) = \{a, c, f\} \). To process the query \( (h|r)(a|bc|def) \), supposing that \( E \) is processed first, the following logical operator is \( (h|r) \rightarrow E((a, b, d)) \) corresponding physical operator list is Load\( (\{h|r\}) \); Neighbor\( (\{h|r\}, \{a, b, d\}) \); SingleLink\( (\{h|r\}, \{a, b, d\}, \{a, c, f\}) \).

For SingleLink, it is supposed that the links between a node and b node as well as the links between b nodes and c nodes have been built. For DoubleLink, it is supposed that the links between a nodes and b nodes, b nodes and c nodes, c nodes and d nodes have been built. For ClosureLink, it is supposed the links between a nodes and b nodes have been built.

The relationships between logical operators and physical operators are shown in Table 3. In Table 3, for closure, the function Link depends on the form of \( E_1 \) and \( E_2 \). If both \( E_1 \) and \( E_2 \) are single labels, this link operator is unnecessary; if only one of \( E_1 \) and \( E_2 \) is a complex regular expression and the other is a single label, the operator is SingleLink; if both of \( E_1 \) and \( E_2 \) are a complex regular expressions, corresponding operator is DoubleLink.

We use an example to illustrate the intermediate results and the physical operator execution.

Example 2. In this example, we show the execution of physical operators for the query and logical operator execution.
TABLE 1
The Table of Logical Operators

| Operator | Logical Operator | Semantics |
|----------|-----------------|-----------|
| concentration | $a - b$ | directly concentration between an $a$ node and a $b$ node |
| | $a - E(b)$ | concentration between a $a$ node and the $b$ node in the result of a RE $E$ |
| | $E(b) - a$ | concentration between the $b$ node in result of a RE and a $a$ node |
| | $E_1(a) - E_2(b)$ | concentration of the results of $E_1$ and $E_2$ on the $a$ and $b$ nodes |
| | $E(E_1(a) - E_2(b))^+$ | the closure of a RE where head is an $a$ node as the tail of $E_1$, tail is a $b$ node as the head of $E_2$, where $E_1$ and $E_2$ have been processed |
| | $a^+$ | the closure of a single label $a$ |

TABLE 2
Physical Operators

| Operator | Meaning |
|----------|---------|
| Load(a) | Load the nodes with label $a$ |
| Neighbor(a, b) | Obtain $b$ neighbors of each $a$ nodes and link each $a$ node with its $b$ neighbor |
| SingleLink(a, b, c) | Link the $c$ neighbors of each node with label $b$ to all its $a$ neighbors |
| DoubleLink(a, b, c, d) | Link each pair of $a$ and $b$ nodes where each $a$ node is a neighbor of a $b$ node $N_a$, each $d$ node is a neighbor of a $c$ node $N_c$, and $N_b$ links $N_c$ node |
| ClosureLink(a, b) | Link each $a$ node $N_a$ with all its $b$ descendants with each such $N_b$ exits a path $N_aN_bN_1N_2\cdots N_nN_aN_0$ between them, where $N_a$, $N_b$, $N_1$, $N_2$, $\cdots$, $N_n$, $N_0$ have label $b$ |
| SelfLink(a, a1, a2) | generate a copy of the results of $a$ and link the nodes in two sets referring to the same nodes in the graph |

TABLE 3
The Relationships between Logical operators and Physical Operators

| Logical Operator | Physical Operator List |
|------------------|-----------------------|
| $a \rightarrow b$ | Load(a); Neighbor(a, b) |
| $a \rightarrow E(b)$ | Load(a); Neighbor(a, b); SingleLink(a, b, tail(E)) |
| $a \rightarrow E(b)$ | Load(b); Neighbor(b, a) |
| $E(b) \rightarrow a$ | Neighbor(b, a); SingleLink(head(E), b, a) |
| $E(b) \rightarrow a$ | Load(b); Neighbor(a, b); SingleLink(head(E), b, a) |
| $E_1(a) \rightarrow E_2(b)$ | Neighbors(a, b); DoubleLink(head(E1), a, b, tail(E2)) |
| $E(E_1(a) - E_2(b))^+$ | ClosureLink(E1); Link(head(E1), a, b, tail(E2)) |
| $a^+$ | Load(a), SelfLink(a, a1, a2); ClosureLink(a, a1, a2) |

TABLE 4
The Logical Operators and Physical Operators for the Example

| LID | Logical operator | PID | Physical Operator |
|-----|------------------|-----|-------------------|
| L1  | $a \rightarrow b$ | $b \rightarrow (b, c)$ | $P_{11}$ | Load(a); Neighbor(a, {b, c}) |
| L2  | $a(b|c)|b) \rightarrow e$ | $P_{21}$ | Neighbor(b, e) |
| L3  | $(c \rightarrow d)^+ (E_1(c), d)$ | $P_{31}$ | ClosureLink(c, d) |
| L4  | $E_2(d) \rightarrow f$ | $P_{41}$ | Neighbor(d, f) |
| L5  | $E_3(e, f) \rightarrow g$ | $P_{51}$ | Neighbor(e, f, g) |

The intermediate results for the operators are shown in Figure 2. For the second logical operator $a(b|c)|(b) \rightarrow e$, at first, the $e$ neighbor of $b$ nodes are obtained and then SingleLink operator is performed to connect $a$ node with $e$ nodes. For the logical operator $(c \rightarrow d)^+ (E_1(c), d)$, the first physical operator is ClosureLink(c, d), the $c$ nodes as the input include $c_1$ and $c_2$, but $c_3$ is not the input of this operator. The results of the physical operator are $(c_1, d_1), (c_1, d_2), (c_1, d_3), (c_2, d_1), (c_2, d_2)$ and $(c_2, d_3)$. Then $a_1$ is connected with $d_1, d_2$ and $d_3$.

The efficient implementation algorithms of these physical operators will be discussed in the next section.

3.3 Logical Operator Generation

We explain the conversion from a regular expression to logical operators. Such generation is implemented using postfix expression.

Any infix regular expression can be easily rewritten into a postfix expression. For example, given infix expression $ab - ((b - c) \cdot ((d|e) - f)) - (g|h)$, its postfix expression is $abc - def|\cdot|\cdot|\cdot g|h|$. Given a postfix expression, we can generate its logical operators using a tri-column stack in the schema of (operand, prefix, postfix). The operand is the operator, while prefix/postfix column stores the prefix/postfix character set of current stack frame. The algorithm yields a logical operator when meeting concatenation operator “$\cdot$”, concatenating the postfix of the first operand and the prefix of the second operand. It merges prefixes and postfixes when meeting “or” operator “$|$”, merging the prefixes/postfixes of two operands.

The logical operator generation algorithm consists of three steps:

- Scan the postfix expression from left to right;
- If current character is an operand, then push it to the stack;
- If current character is an operator, pop two operands, doing calculation and push the result back.

4 IMPLEMENTATION ALGORITHMS OF PHYSICAL OPERATORS

In this section, we will discuss the implementation algorithms of physical operators in Section 3.2. For the convenience of discussions, we introduce centralized algorithms and distributed algorithms in Section 4.1 and Section 4.2 respectively.
4.1 Centralized Algorithms

This section proposes the implementation algorithms for physical operators in centralized environment. Note that the algorithms are implemented with a lightweight index or in absence of index. The description of the algorithms in this section could be easily implemented on a database with join operators.

To simplify the algorithm descriptions, the symbols and functions used in the algorithm descriptions are shown in Table 5.

Since the implementation of SelfLink is straightforward, we focus on other operators.

Load: The implementation of the physical operator Load is simply load the nodes with label in S(a) from the graph. With the index that retrieve the nodes according to the label, this operator is performed in time linear to the node number. Such index is called label index, as an inverted list that retrieves the id sets according to the given label, and the size is O(n).

Neighbor: The operator of Neighbor is implemented by load each neighbor with the label in S(b) for each node in R_l (l ∈ S(a)) and link them. At last, the nodes with label in R_l without any neighbor with node in S(b) are filtered. For the efficiency issue, the implementation of this operator requires accessing the neighbors of corresponding nodes and merge their ids with the id set in corresponding entry in the label index or corresponding intermediate node sets to avoid accessing the labels of nodes.

The implementation of the operator Neighbor is shown in Algorithm 1.

Algorithm 1 Neighbor(a, b)

for each r in S(a) do
  for each n in R_r do
    for each r2 in S(b) do
      load all neighbors of n with label r and N_{n,r}
      for each node n2 in N_{n,r} do
        link(n, n2, r, r2)

SingleLink and DoubleLink: As described in Table 2, the goal of SingleLink and DoubleLink is to link nodes in the head and tail in a path. In order to reduce redundancy computation, we apply BFS strategy. As shown in Algorithm 2, at first the nodes in R_{r2} (r2 ∈ S(b)) linked by nodes in R_{r1} (r1 ∈ S(a)) are collected in M. (Line 3-Line 5). Then for each node n’ in M, the nodes with nodes in R_{r3} (r3 ∈ S(c)) linked by n’ (Line 9-Line 11). At last, the nodes in each R_{r1} and each R_{r3} are filtered based on their links (Line 10 and Line 12). As shown in Algorithm 3, the implementation of DoubleLink is similar as SingleLink but with an additional step of collection neighbors in R_{r3} (r3 ∈ S(c)) for each node in M (Line 7-Line 9).

Algorithm 2 SingleLink(a, b, c)
1: for each r1 in S(a) do
2:   for each n in R_{r1} do
3:     M=∅
4:     for each r2 in S(b) do
5:       M=M ∪ N_{n,r2}
6:     for each n’ in M do
7:       for each r3 in S(c) do
8:         for each n2 in N_{n’,r3} do
9:           link(n1, n2, r1, r3)
10:          Filter(r1, S(c))
11:        for each r in S(c) do
12:          Filter(r, S(a))

For example, to perform the operator SingleLink(a, \{e,f\}, g) on the result in Figure 2(b) at first, e and f nodes connected to a node are obtained. The set of e and f nodes connected to a1 is S1={e1, f1, f2, f3}. As the result, the g nodes as the neighbors in the nodes in S1 are obtained. Then, the set \{g1,g2,g3\} is obtained and a1 is linked with g1,g2 and g3.

Algorithm 3 DoubleLink(a, b, c, d)
1: for each r1 in S(a) do
2:   for each n in R_{r1} do
3:     M=∅
4:     for each r2 in S(b) do
5:       M=M ∪ N_{n,r2}
6:     for each n’ in M do
7:       for each r3 in S(c) do
8:         for each n2 in N_{n’,r3} do
9:           link(n1, n2, r1, r3)
10:          Filter(r1, S(c))
11:        for each r in S(c) do
12:          Filter(r, S(a))
13:         for each r2 in S(b) do
14:           Filter(r2, S(a))
15:          for each r3 in S(d) do
16:            Filter(r3, S(d))
17:          Filter(r, S(a))

ClosureLink: The pseudo code of the implementation algorithm of ClosureLink is shown in Algorithm 4. The idea is to obtain the nodes to connect with ClosureSearch. During the search, to avoid repeated search, we use a hash table H to maintain a with all descendants searched. For each vertex n matching a, a descendant set M_n is generated for all corresponding b vertices by ClosureSearch.

To avoid redundant search, ClosureSearch is implemented in the combination of DFS and BFS. For each vertex n with label a, BFS is applied to collect all its b neighbors in H_b and add all the neighbors to the a ancestors of n in S (Line 16-Line 19). Then, for each node n’ in H_b, DFS
is applied to obtain all its a ancestors and b ancestors. ClosureSearch is invoked for each unvisited a neighbor \( n'' \) of \( n' \) (Line 23). If \( n'' \) has been visited, the corresponding b nodes are copied to the ancestors (Line 27 and Line 32).

For example, to perform the ClosureLink(\( c, d \)) on the c node set \( \{c_1, c_2\} \), \( c_1 \) is to be processed at first and pushed to the stack \( S \). \( d_1 \) is added to \( H_b \) and \( M_{c_1} \). Then, the traversal starts from \( d_1 \) and \( c_3 \) is obtained. \( c_3 \) is pushed to \( S \). The traversal starting from \( c_3 \) obtains \( d_2 \) and \( d_3 \). \( d_2 \) and \( d_3 \) are added to \( M_{c_1} \) and \( M_{c_3} \). When the search from \( c_1 \) is accomplished, \( c_1 \) is linked to all vertices in \( M_{c_1} \). After that, since \( c_3 \) has been visited, \( c_3 \) is linked to all vertices in \( M_{c_3} \).

Note that descendants may have duplications. It is caused by repeated segments in the graph matching the clause in the RE. Such duplications could be avoided by adding links to sets instead of copying sets (in Line 29 and Line 32). From the experiments, such cases seldom occur.

\textbf{Algorithm 4 ClosureLink(} \( a, b \)\textbf{)}

1. \( H=\emptyset \)
2. \( \text{for each } l \text{ in } L(a) \) do
3. \( \quad \text{for each } n \text{ in } R_l \) do \( \text{do} \)
4. \( \quad \quad \text{if } n \text{ is not in } H \text{ then} \)
5. \( \quad \quad \quad M_n = \emptyset \)
6. \( \quad \quad \quad S = \emptyset \)
7. \( \quad \quad \text{ClosureSearch}(n, l, a, b, M_n, H, S) \)
8. \( \quad \quad \text{add } n \text{ to } H \)
9. \( \quad \text{for each } k \text{ in } M_n \) do \( \text{do} \)
10. \( \quad \quad \text{link}(n, k, r, \text{label}(k)) \)
11. \( \text{ClosureSearch}(n, l, a, b, M, H_a, S) \)
12. \( \text{Push}(S, n) \)
13. \( H_b = \emptyset \)
14. \( \text{for each } l \text{ in } S(b) \) do \( \text{do} \)
15. \( \quad \text{for each } j \text{ in } N_{n,l} \) do \( \text{do} \)
16. \( \quad \quad \text{for each } k \text{ in } S \) do \( \text{do} \)
17. \( \quad \quad \quad \text{if } j \in R_l \text{ then} \)
18. \( \quad \quad \quad \quad \text{add } j \text{ to } M_k \)
19. \( \quad \quad \quad \text{add } j \text{ to } H_k \)
20. \( \quad \text{for each } n' \text{ in } H_k \) do \( \text{do} \)
21. \( \quad \quad \text{for each } l' \text{ in } S(a) \) do \( \text{do} \)
22. \( \quad \quad \quad \text{if } n'' \notin H_a \text{ and } n'' \notin S \text{ then} \)
23. \( \quad \quad \quad \quad \text{ClosureSearch}(n'', l', a, b, M_{n''}, H_a, S) \)
24. \( \quad \quad \quad \quad \text{add } n'' \text{ to } H_a \)
25. \( \quad \quad \quad \text{else if } n'' \in H_a \text{ then} \)
26. \( \quad \quad \quad \quad \text{for each } r \text{ in } S \) do \( \text{do} \)
27. \( \quad \quad \quad \quad \quad M_r = M_r \cup M_{n''} \)
28. \( \quad \quad \quad \text{else} \)
29. \( \quad \quad \quad \quad \text{for } r = \text{top}(S) \text{ to prior}(n'') \) do \( \text{do} \)
30. \( \quad \quad \quad \quad \quad M_r = M_r \cup M_{n''} \)
31. \( \quad \quad \text{POP}(S) \)

Note that during the operator processing, the original graph is not modified. When a vertex \( n \) in the graph is obtained, a stub node for \( n \) is constructed in the intermediate result. During traversal, the adding of a link between \( n \) and other node \( m \) is implemented by adding such link between corresponding stub nodes.

\textbf{Complexity Analysis} For a graph with \( n \) vertices and \( m \) edges, obviously, in the worst case, the time complexity of operator Load and SelfLink is \( O(n) \) since at most each vertex is accessed only once. In the worst case, for each vertex \( v \), the operator Neighbor will access at most \( n \) vertices. Then the time complexity of Neighbor is \( O(n^2) \).

With BFS search strategy, for each step, each dummy vertex is processed only once and for each vertex the vertices with special labels linked to it will be visited. Therefore, the time complexities of SingleLink and DoubleLink are both \( O(n^2) \).

\section{4.2 Distributed Algorithms}

To handle very large graphs, it naturally adopts a distributed platform. In order to process regular expression query on large distributed graphs efficiently, our solution provides efficient physical operator implementation algorithms on the distributed platform to minimize the communication cost. From the aspect of graph management, the possible communication cost is caused by traversing from the vertex in one machine to those in another machine. Thus, we should select the distributed graph management platform supporting efficient traversal.

Motivated by this, we adopt infrastructure of Microsoft Graph Engine \( ^{[1]} \) an open-source distributed in-memory graph processing engine which supports traversal efficiently. It is underpinned by a strongly-typed in-memory key-value store and a general-purpose distributed computation engine. The following two core capabilities of Graph Engine make it an ideal platform for handling distributed large graphs with complex data schema: 1) It excels at managing a massive amount of distributed in-memory objects and providing efficient random data access over the distributed data. Fast random access is the key to many graph algorithms. 2) It excels at handling big graph data with complex schema. For example, Graph Engine is serving a Microsoft knowledge graph (about 23 TB) which has thousands of entities types and billions of nodes and edges.

Thus, such platform is suitable for efficient distributed implementation of Neighbor, SingleLink, DoubleLink and ClosureLink.

In distributed environment, for the convenience of processing, each node is stored locally, and only the ids of nodes are transmitted during query processing.

With the consideration of network issues, the Load and SelfLink are only performed locally. However, for other operators, the network communications may be involved.

Clearly, a pretty graph partition among the machines in a cluster could accelerate the processing. However, the graph partition is costly in computation especially for very large graphs. Additionally, in real-world scenarios, the workload

1. \url{http://research.microsoft.com/en-us/projects/graphengine/}
on a graph distributed in a cluster may contain various operations instead of only regular expression query processing. Thus, it is difficult to choose a proper partition criteria. To make our approach suitable for real applications, we suppose that the graph is partitioned among machines randomly without any sophisticated graph partition strategy. Basic Operations From the implementation strategy, the basic operations related to two nodes are fetching a neighbor from a node and connecting two nodes. Therefore, the queries are processed in distributed environment by two network primitives according to the two operations for the nodes distributed in different machines in the network.

- GetNeighbor(n, t, t_r, t_a) obtains the neighbors with tag t_r for the nodes n with tag t_a.
- AddLink(u, v, a, b) connects u and v with u as v’s neighbor and v as u’s neighbor.

In order to save the communication cost and network bandwidth, the distributed implementation has two strategies. One is to load node ids instead of all information of nodes. The other is to load and revise link in batch. The pseudo code of the implementations of GetNeighbor and AddLink, respectively. The input of GetNeighbor is a table in schema (id, r1, r2), with the same semantics as that of GetNeighbor. The returned results are in schema (pid, plabel, cid, clabel), where pid is the id of the input tuple, plabel is its label, cid is the id of the obtained neighbor, and clabel is the label of the neighbor. The input of AddLink is in schema (pid, plabel, cid, clabel), where pid is the id of the input tuple, plabel is its label, cid is the id of the neighbor, and clabel is the label of this neighbor to link.

In the intermediate results, for each node, the links to neighbors are the ids of the node. In each machine, to access the real node efficiently, a hash table is maintained to map the id to the real node. And in the whole system, the id of computer for each node n is encoded in the id of n.

**Example 3.** Consider the process of Q1 in the graph in Figure 1. It is supposed that there are five computers M1, M2, M3, M4, and M5 in the cluster. The partition of nodes in the cluster is shown in Figure 2. For the execution physical operator SingleLink(a, {e, f}, g) on the intermediate results shown in Figure 2 at first, starting from a1, the network primitives of GetNeighbor(a1, {v1,f1}), g) and GetNeighbor(a1, {f2,f3}, g) are generated. Then, they are sent to M2 and M3, respectively. After M2 and M3 execute the primitives. The ids of g1, g2, and g3 are returned to M1. a1 is linked with g1, g2, and g3. With these ids, the locations of g1, g2, and g3 are known. With these locations, the primitives of link, AddLink(a1, {g1, g2}), a, g) and AddLink(a1, {g3}, a, g), are generated and sent to M4 and M5, respectively. With these primitives, in M4, g1 and g2 are linked with a1, and in M5, g3 is linked with a1.

**Example 4.** To process ClosureLink(c, d) on the graph in Figure 1 with graph partition in Figure 2. At first, the search starts from c1 and c2 in M1 and M2, respectively. As the result, d1 and d2 are obtained. The primitives GetNeighbor(d1, c, d) and GetNeighbor(d2, c, d) then request c nodes from M3. They are processed in M3 in batch. From the information stored in c3, the ids of d1, d2, and d3 are sent to M1 and M2, respectively. In M1, c1 is linked with d2 and d3, and the primitives AddLink(c1, d2, c, d) and AddLink(c1, d3, c, d) are generated and sent to M2 and M3, respectively. In M2, c2 is

---

**Algorithm 5 DistributedSingleLink(a, b, c)**

```plaintext
for each r1 in S(a) do
    for each n1 in R_T1 do
        M=∅
        for each r2 in S(b) do
            for each n2 in N_n1,r2 do
                add (r1, r2, n1, n2) to T1
        for each t in T1 do
            add (t[pid], t[plabel]) to M_t[pid, t[plabel]]
        for each r in S(c) do
            add (r[cid], r[clabel], t to T2)
        R_T2=BachGetNeighbor(T2)
    for each r in T2 do
        for each r in M_t[pid, t[plabel]] do
            add (r[cid], r[clabel], t[pid], t[plabel]) to T3
        BatchAddLink(T3)
        for each r1 in S(a) do
            Filter(r1, S(c))
        for each r2 in S(c) do
            Filter(r2, S(a))
```

---

**Figure 3. The distributed Graph**
Algorithm 6 DistributedClosureLink(a, b)
1: $H = \emptyset$
2: for each $l$ in $S(a)$ do
3:   for each $n$ in $R_l$ do
4:     for each $l'$ in $S(b)$ do
5:       if $(n, l, l')$ is not in $H$ then
6:         add $(n, l, l')$ to $H$
7:     for each $n'$ in $N_{n,l'}$ do
8:       add $(n, l, n', l')$ to $R_1$
9:   while $R_1 \neq \emptyset$ do
10:     $N = \emptyset$
11:    for each $t$ in $R_1$ do
12:       add $(t[plabel], t[pid])$ to $M_1[(t[clabel], t[clid])]$
13:       $(t[clabel], t[clid])$ to $N$
14:     for each $t$ in $N$ do
15:       for each $l$ in $S(a)$ do
16:         add $(t[clid], t[clabel], \text{Label}(l))$ to $T_2$
17:     $R_2 = \text{BatchGetNeighbor}(T_2)$
18:     $N = \emptyset$
19:    for each $t$ in $R_2$ do
20:       add $M_2(t[plabel], t[pid])$ to $M_2[(t[clabel], t[clid])]$
21:       $(t[clabel], t[clid])$ to $N$
22:     for each $t$ in $N$ do
23:       for each $l'$ in $S(b)$ do
24:         if $(n, l, l')$ is not in $H$ then
25:           add $(n, l, l')$ to $H$
26:         add $(n, l, l')$ to $T_1$
27:     $R_1 = \text{BatchGetNeighbor}(T_1)$
28:   for each $t$ in $T_1$ do
29:     for each $r$ in $M_2[(t[plabel], t[pid])]$ do
30:       add $(r[id], r[plabel], t[clabel], t[clid])$ to $T_3$
31:     $\text{BatchAddLink}(T_3)$
32:   for each $r_1$ in $S(a)$ do
33:     $\text{Filter}(r_1, S(b))$
34:   for each $r_2$ in $S(b)$ do
35:     $\text{Filter}(r_2, S(a))$

linked with $d_1$ and $d_2$, and the primitive $\text{AddLink}(c_2, d_1, c, d)$ and $\text{AddLink}(c_2, d_3, c, d)$ is generated. They are sent to $M_1$ and $M_2$, respectively.

Then, in $M_1$, $d_1$ is connected with $c_3$; in $M_2$, $d_2$ is linked with $c_1$; in $M_3$, $c_1$ and $c_2$ are linked with $d_3$. From the aspect of $M_1$, the search then starts from $d_2$ with $c_1 d_1 c_3 d_2$ as the traversal path. $c_2$ is obtained. Since $c_2$ has no other $d$ neighbor. The traversal in $M_1$ halts. Similar process is performed in $M_2$.

5 QUERY OPTIMIZATION
As discussed in Section 3, each RE query may have different processing orders and directions of the operators. The execution order may affect the efficiency of query processing. For example, even for the simple query $abc$, it is supposed that each of $1G$ nodes with tag $a$ is linked with one different $b$ node and only one of the $b$ nodes is connected with a $c$ node. If the query is executed in the direction from $a$ to $c$, more than $2G$ nodes are accessed during search, while only two nodes are accessed if the query is executed from $c$ to $a$.

Motivated by this, in this section, we develop the query optimization strategy for RE queries. As the base of query optimization, we propose the cost estimation methods for operators and the size of intermediate results in Section 5.1

5.1 Estimation
In this section, we propose a cost-based query estimation method for query optimization. Since for a large graph, it is impossible to maintain a global sketch with size super linear to the graph size. Therefore, in our system, only following statistic information is kept, which is independent to the graph size.

1) The number of the nodes with label $l$, denoted by $Num(l)$
2) The average neighbor number of the nodes with label $l$, denoted by $\text{Neighbor}(l)$
3) The probability of a node with label $l_1$ has at least one neighbor with label $l_2$ respectively, denoted by $\text{Pro}(l_1, l_2)$
4) The average number of the neighbor with label $l_2$ of a node with label $l_1$ if it has at least one neighbor with $l_2$, denoted by $\text{TNeighbor}(l_1, l_2)$

Such information is computed by traversing the graph once. In a large graph, the size of $\text{Num}(l)$ and $\text{Neighbor}(l)$ for all labels is linear to the size of the label number $L$. However, the size of $\text{Pro}$ and $\text{TNeighbor}$ is $L^2$. In order to deal with the graph with a large number of labels, we use two thresholds $\epsilon_T$ and $\epsilon_T$. For two labels $l_1$ and $l_2$, if $\text{Pro}(l_1, l_2) < \epsilon_T$, a small value $\delta_T$ is used instead of the real value of $\text{Pro}(l_1, l_2)$. Similarly, if $\text{TNeighbor}(l_1, l_2) < \epsilon_T$, the real value of $\text{TNeighbor}(l_1, l_2)$ is replaced by a small value $\delta_T$ during optimization. Thus, the small size of statistic information is ensured.

Based on above statistics information, we introduce the estimation approach. As discussed in Section 4, one label may exist multiple times in a RE corresponding to different set of intermediate results. During query processing, the size of intermediate results will change. In order to distinguish multiple occurrences of the same label in the optimization, we assign a uniform id for the occurrences of each label and use $L(x)$ to denote the corresponding label of the label with id $x$. We use $I_x$ to denote the set of intermediate results corresponding to $x$. In the remaining part of this section, without explicitly explanation, the variables in the operators and formulas refer to a set of ids instead of labels.

For the convenience of discussions, we discuss the estimation of operators without ‘|’ as the basic version. Then we discuss the extension to the operators with ‘|’

5.1.1 The Estimation of Operators without ‘|’
For the brief of discussion, in the beginning, we focus on the estimation for operators without ‘|’. The subqueries with ‘|’ will be discussed later. It means that each variable refers to single id.

Basic Information The cost estimation is based on the sizes of intermediate results and the number of links between the nodes in two intermediate result sets. Thus, we discuss the estimation of such parameters at first. The symbols used in such estimation are described in Table 1. Then we discuss the size and link number estimation in the following paragraphs. The results are summarized in Table 2 where
TABLE 6
Symbols in Estimation

| Symbol     | Meaning                                                                 |
|------------|-------------------------------------------------------------------------|
| Link(a, b) | the average number nodes in \( I_a \) linked by each node in \( I_b \) after the operator computation |
| size'(a)   | the size of \( I_a \) before the operator computation                   |
| size(a)    | the size of \( I_a \) after the operator computation                    |

the column Link is the number of links between of two sets of intermediate results, and column Size is the numbers of the intermediate results.

Result Size of SingleLink and DoubleLink The estimated size of Load and SelfLink is intuitively the number of corresponding nodes. We use SingleLink as an example to explain the idea of the estimation SingleLink and DoubleLink. For the operator SingleLink, Link(a, b) \( \text{size}(a) \) is the average number of links between \( I_a \) and the nodes in \( I_b \) with at least one node. With Link(b, c) as the average number of links between \( I_b \) and \( I_c \), for each node in \( I_b \), the estimated link number between \( I_a \) and \( I_c \) is multiplied by Link(b, c) to compute the average number of links between each node in \( I_a \) and the nodes in \( I_c \) through some nodes in \( I_b \). Since each node \( n \) in \( I_a \) is connected with a node in \( I_b \), the probability of \( n \) links with at least a node in \( I_b \) through the node in \( I_b \) is the probability that a node in \( I_b \) links to at least a node in \( I_a \). The estimation of the size of \( I_c \) after the operator execution is similar.

Result Size of ClosureLink In the formula of estimated links in ClosureLink(a, b), \( t \) means the average number of link between each \( b \) node \( n_b \) in \( I_b \) and the node in \( I_b \) through the path \( n_b - n_a - n'_b \), where \( n_a \in I_a \) and \( n'_b \in I_b \). In \( t \), \( \text{Pro}(b, a)T_{\text{Neighbor}}(b, a) \) is the average number of the neighbors with label \( L(a) \) of each node of \( I_a \) and \( \text{Pro}(b, a)T_{\text{Neighbor}}(b, a) \) is the average number of nodes with label \( L(a) \) of each node of \( I_b \). \( \text{Pro}(a, b)\text{size}'(a, b) \) is the average number of nodes with label \( L(b) \) linked by each node with label \( L(a) \) in the intermediate result. Hence, \( t \) represents the average number of nodes in \( I_b \) of each node in \( I_a \) with a path \( a - b \) between them. Since the computation of ClosureLink requires the traversal of the paths in label form \( a - b - a - \cdots - b \) with the all possible length, the average number of links in the results is estimated as \( \text{link}'(a, b) + \text{link}'(b, a) + \cdots + \text{link}'(b, a)^k + \cdots \). When \( t \leq 1 \), since the maximal number of links between each node in \( I_a \) and the nodes in \( I_b \) is \( \text{size}'(b) \), the estimation of average node number in \( b \) lined with each node in \( I_a \) is \( \text{size}'(b) \). Since before ClosureLink operation, each node in \( I_a \) has been linked with at least one node in \( I_b \) and each node in \( I_b \) has been linked with at least one node in \( I_a \), the size of \( I_a \) and \( I_b \) are not changed after the operation.

With this information, the cost estimation formulas of physical operators are shown in Table 6.

Cost of SingleLink and DoubleLink Since the operators Load and SelfLink access each node in the candidate only once, their costs are the same as the number of candidates. The operator SingleLink has two phases, the first phase collects the nodes in \( I_b \) for the nodes in \( I_a \) with cost \( \text{size}'(a) \cdot \text{Link}(a, b) \), and the second phase collects the nodes in \( I_a \) linked with nodes in \( I_b \) with cost \( \text{size}'(a) \cdot \text{Link}(a, b) \cdot \text{Link}(b, c) \). The estimation for DoubleLink is similar.

Cost of ClosureLink As shown in Algorithm 6, the operator ClosureLink has multiple phases, each phase has two steps. The first is to obtain the nodes in \( I_b \) that is connected with the nodes in \( I_a \) which require to access the neighbors of each node in \( I_b \). From the aspect of starting from a single node in \( I_a \), in the first step, the cost is \( \text{Link}'(a, b) \). In the first step of the next phase, the number of the starting nodes in \( I_a \) is \( \text{size}'(b) \cdot \text{Pro}(b, a) \cdot \text{size}'(a) \cdot \text{Num}(a) \), since the number of all nodes with label \( L(b) \) in the second step of the last phase is \( \text{size}'(b) \). The cost of the first step in the second phase is \( \text{size}'(b) \cdot \text{Pro}(b, a) \cdot \text{size}'(a) \cdot \text{Num}(a) \).

Therefore, the number of nodes in \( I_a \) as the input of the first step of the third phase is

\[
\text{size}'(b) \cdot \text{Pro}(b, a) \cdot \text{size}'(a) \cdot \text{Num}(a) = \text{size}'(b) \cdot \text{Pro}(b, a) \cdot \text{size}'(a) \cdot \text{Num}(a)
\]

Therefore the cost of the first step of the \( k \)th phase \((k > 1)\) is \( \text{size}'(b)^{k-1} \cdot \text{size}'(a) \cdot \text{Link}'(a, b) \), where \( r = \text{Pro}(b, a) \cdot \text{size}'(a) \cdot \text{Num}(a) \).

With the consideration that all \( b \) nodes are accessed only once, the cost of the second step of all the phases is \( I(b) \cdot T_{\text{Neighbor}}(b, a) \). For the case that \( r \) is larger than 1, the cost is estimated as the maximum cost of search all possible nodes with labels \( L(a) \) and \( L(b) \).

5.1.2 The Estimation of Operators with ‘|’

Above estimation focuses on the operators without ‘|’ in the input. Then we discuss the estimation for the cases with ‘|’, where each variable in the operator may refer to multiple labels.

For the operators of Load and SelfLink, the cost and size is the sum the cost and result size of all labels.

Since the operators of Neighbor, SingleLink and DoubleLink with variables referring to multiple labels can be considered as the execution of a series of operators with the same type and variables referring to single label, their costs and link numbers can be estimated as the sum of the cost and the number of links of all possible combination of the Cartesian production of the candidate sets, respectively.

For example, the results of the operator SingleLink(a_1|· · ·|a_m, b_1|· · ·|b_n, c_1|· · ·|c_k) is considered as a set of operators with each one SingleLink(a_i, b_j, c_l) where \( i \in [1, m], j \in [n], l \in [1, k] \). Therefore, its cost is estimated as follows.

\[
m \sum_{i=1}^{m} \sum_{j=1}^{n} \text{Link}(a_i, b_j) \sum_{l=1}^{k} \text{Link}(b_j, c_l)
\]

Its number of links is estimated as

\[
\text{Link}(a_i, c_l) = \sum_{j=1}^{n} \text{Link}(a_i, b_j) \cdot \text{Pro}(b_j, c_l) \cdot \text{Link}(b_j, c_l) \cdot \frac{\text{size}'(c_l)}{\text{Num}(c_l)}
\]
For $a_i$, 

$$\text{size}(a_i) = \max \{ \text{size}'(a_i), \sum_{j=1}^{n} l \times \text{size}(b_j, c_j) \}$$

For ClosureLink, since the estimation involves multiple nodes, we use a matrix to represent the estimation. For example, for the operator ClosureLink($a|b|c|d$), the patterns of a possible matched path in length 4 may be $acac$, $acad$, $acbd$, etc. The estimation of the cost of traversing such path require the consideration of all possible cases. To enumerate all the cases for the formula of ClosureLink cost estimation, we use matrices to represent the links, probabilities and the number of neighbors between labels and use the computation on the matrix to compute the costs and the links. The matrices used in the estimation are listed as follows. $A$ is an $m \times m$ matrix, in which $A_{ij} = \text{size}'(a_i)$, $A_{ij} = 0$ (i.e., there is no link from $a_i$ to $a_j$); $B$ is an $n \times n$ matrix, in which $B_{ij} = \text{size}'(b_i)$, $B_{ij} = 0$ (i.e., there is no link from $b_i$ to $b_j$); $P$ is an $n \times m$ matrix with each entry $P_{ij} = \text{Pro}(b_i, a_j) \times \text{size}(a_i)$; $R$ is an $m \times n$ matrix with each entry $R_{ij} = \text{size}(b_j) / \text{size}(a_i)$; $L$ is an $m \times n$ matrix with each entry $L_{ij} = \text{Link}'(a_i, b_j)$.

Once the matrices are computed, we use the following formula with $R$ as input to the formula of ClosureLink cost estimation for $a_i$.

$$\text{size}(a_i) = \text{size}(a_i) \times \text{Pro}(b_i, a_j) \times \text{size}(a_i) \times \text{Link}'(a_i, b_j)$$

When the cost of the first step in the second phase is estimated as $BPTP$ and the cost is estimated as the matrix $BPTPL$. In summary, the cost for ClosureLink($a_1|b_2|\cdots|a_m|b_1|b_2|\cdots|b_n$) is estimated as follows.

$$\sum_{1 \leq i \leq m, 1 \leq j \leq n} F_{ij} + \sum_{1 \leq i \leq n, 1 \leq j \leq n} E_{ij}$$

where $F = AL$ and $E = BPL + BPRPL + \cdots$. Note that the number of items in the computation formula of $R$ is smaller than the diameter of the graph. To accelerate the processing, we use a constant $r$, only the sum of the first $r$ items is used.

The link number of ClosureLink is estimated by phases. The link number generated in the first phases is $L$, each entry $L_{ij}$ of which represents the number of original link between $I_{a_i}$ and $I_{b_j}$. Following the link number estimation formula for ClosureLink in Table 7, the links generated in the second phase is estimated as the matrix $LT_1T_2$ with each entry $r_{ij}$ as the number of additional links from $I_{a_i}$ to $I_{b_j}$.

$$\sum_{1 \leq i \leq m, 1 \leq j \leq n} F_{ij} + \sum_{1 \leq i \leq n, 1 \leq j \leq n} E_{ij}$$

where $F = AL$ and $E = BPL + BPRPL + \cdots$. Note that the number of items in the computation formula of $R$ is smaller than the diameter of the graph. To accelerate the processing, we use a constant $r$, only the sum of the first $r$ items is used.

5.2 Query Optimization Algorithm

To obtain the optimal query plan for a query, we develop the query optimization algorithm for regular expression query based on the cost model introduced in the last section. Since each logical plan operator corresponding to a fixed series of physical operators. We focus on logical operators in this section.

A straightforward method for query optimization is to enumerate all possible query plan and choose the cheapest one. For example, the search space for query $abcd$ is shown in Figure 4, where each state corresponds to an operator and each state may be reduced to $\rightarrow$ or $\leftarrow$. For each state, the execution order and direction should be considered. For example, for the
state $a - bcd(b)$, the execution order of $b - c$ and $c - d$ is to be determined and the direction of the operator $a - bcd(b)$ is also to be determined.

Obviously, when the regular expression gets complex, if such brute-force method is applied, the query optimization is costly. It is observed that the search space of possible query plans has overlapping space for subqueries. In the above example, state $a - bcd(b)$ and state $abc(e) - d$ share the same search space $b - c$. Then the direction of $b - c$ will be determined twice. Based on this observation, we propose a dynamic programming algorithm for query optimization for regular expression. To simplify the discussion, we will discuss the processing of the regular expression with only simple clause with at most one concentration operation in a closure without recursion closure.

At first, the operations in a regular expression $E$ is modeled as an operation graph $G_E$ based on their adjacent relationship. Each operation corresponds to a vertex in the operation graph. If two operations are adjacent, an edge is added to their corresponding vertices in the operation graph.

Each clause $C$ in $E$ corresponds to a subgraph $G_C$ in the $G_E$. Each possible query plan in $E$ corresponds to a spanning tree in $G_E$. Similarly, each possible query plan for a clause $C$ corresponds to a spanning tree in $G_C$. Note that this problem cannot be solved trivially as the cost of the query plan corresponding to a spanning tree is not the sum of the cost of each edge. The recursion function of the dynamic programming is as follows where $Cost_{G_C - u}$ represents the sum of cost of the optimal plans for the clauses corresponding to $Cost_{G_C - u}$ and $Cost(u|G_C)$ is the optimal cost of operator corresponding to $u$ after the optimal query plan for $G_C$ has been executed. $Cost(u|G_C)$ can be computed from two possible direction of $u$ after the processing of $G_C$:

$$Cost_{G_C} = \min_{u \in V_C} \{ Cost_{G_C - u} + Cost(u|G_C) \}$$

Based the recursion function, the pseudo-code of query optimization is shown in Algorithm 7. During processing, a hash table with the key as the bitmap for the vertices in the subgraph is maintained to map the subset of vertices of $G_E$ to their corresponding optimal cost. A hash table $R$ maps each subset of vertices $V$ of $G_E$ to the chosen operator as the last operator for processing the induced graph $G_E[V]$, which consists of $V$ and all edges with endpoints contained in $T$.

We use an example to illustrate the process of the algorithm.

**Example 5.** During the optimization for query $a(be|(cd)^*f)g$, the basic operator include (1) $a(b|c)$; (2)$bc$; (3)$(cd)^*$; (4)$df$; (5)$(e|f)g$. The operation graph is shown in Figure 5. In this figure, each node represents an operator, and each edge represents the concatenation relationship between its vertices. The states and phases of the dynamic programming is shown in Figure 6 where each state corresponds to a set of nodes in the operation graph and a subquery. For each state, the node set is represented as a list of numbers in the above and the corresponding subquery is the string below the numbers. To distinguish the concentration of two adjacent symbols and adjacent symbols without relationship, we use ‘-‘ to represent concentration. The states of higher-level phases are generated with related lower-level phases. The plan corresponding to the state for subquery $a - (b - e)(c - d)^+f$ is chosen from four possible plans $\{E_1=a(b|(cd)^+f),\ E_2=(E_1b-e)\ (|E_2=(E_1b-e)^*(a-(b-c))-d|),\ E_3=(E_3b-e)^*(a-(b-c))-d|,\ E_4=(E_4b-e)^*(a-(b-c))-d|\}$. In each binary, the first entry is the subqueries to be executed first the plan corresponding to which have been com-
To test the impact of query
on the independent assumption in the estimation strategy, we used to our cost model, if a clause is in a closure, except the may match paths with various lengths. Note that according
are that for large graph. Only local statistic information is
during the processing.

tor. Similarity, during the optimization of the whole query, we
such closures are process from inner to
for the whole query. When a query has multiple levels of
plan for
is considered as an operator. Additionally, only a share of candidates corre-
expression approximately. The instable cases are caused by
from other regular expressions with the same length, the
labels and the wildcards are not the same in each regular
expression features, the last type of the queries (Random)
for queries with complex structure and various regular
expression is O(|V|^2).
To accelerate the query optimization, the clause
a_1|a_2|⋯|a_n, where each a_i is a label, is considered as single
label during the processing.
The difficulty in the optimization of recursive closure
is that for large graph. Only local statistic information is
maintained and insufficient for recursive closure, which
may match paths with various lengths. Note that according
to our cost model, if a clause is in a closure, except the
first and the last operator in the clause, the operators will
not affected by the operators outside the clause. Based
on the independent assumption in the estimation strategy,
for the operator ClosureLink(a, b), its cost is proportional
with size(\|a\|) which is the only operator affected by other
operators. Additionally, only a candidates correspond-
ning to the head and tail of the clause in closure will
be affected by the previous operators. For example, for the
query \(a(bcd)^+\), if the subquery \(ab\) is executed before the
closure, the execution number of the operation matching
the path \(bcd\) will be not only determined by the number
of \(b\) in the results matching \(ab\) but also affected by the number
of \(b\) neighbors of \(d\) nodes.
Therefore, for the optimization of closure in a query, we
generate the query plan for the clause in the closure before
the optimization for the whole query. Such plan is used as an
operator without the modification during the optimization
for the whole query. When a query has multiple levels of
recursion closure, such closures are process from inner to
outer recursion.

For example, for the query \(a(b(cdf)^+e)^+g\), the query
plan for \(cdf\) is generated at first. Then the query plan
for \((cdf)^+e\) is generated with the plan for \(cdf\) as an
operator. Similarity, during the optimization of the whole query,
\((cdf)^+e\) is considered as an operator.

6 Experiments
To verify the efficiency of the proposed approaches, we
conduct extensive experiments. In this section, we propose the
experimental results and analysis.

6.1 Experimental Setting
We conduct the experiments on a cluster consisting of 32
servers. Each server has 72GB DDR3 RAM and two 2.67
GHz Intel Xeon X5660 CPU. Each CPU has 6 cores and
12 threads. The network adapters is Broadcom BCN5709C
NetXtreme GigE. Each server’s operating system is Win-
dows Server 2008 R2 Enterprise with service pack 1.
Our code was written with C# and compiled by .NET
Framework 4.5. All the experiments are run on Microsoft
GraphEngine.

We use both real and synthetic data to test our sys-
tem. We design three kinds of queries for each graph to
test the performance of algorithms with various kinds
of queries. The first type of the queries (Hand) is generated
by randomly handwriting some complicated regular expres-
sion as the query pattern with specific semantics. To test
the performance for the queries with at least one result,
the second type of the queries (BFS) is generated by BFS
traversal from a randomly chosen node, and the first \(N\)
odes are kept as the query patent. To test the performance
for queries with complex structure and various regular
expression features, the last type of the queries (Random)
is generated by randomly adding labels and wildcards in
the regular expression.

Each label in the regular expression is chosen from the
label set of the graph. For each kind of queries, we generate
five kinds of regular expressions with lengths 5, 10, 15, 20, 25
respectively, and generate five different regular expression
as query pattern for each length. For each query, we execute
10 times and record its average execution time.

6.2 Experimental Results
6.2.1 Experimental Results on Real Data
For real data, We use freebase data set. It is a large collabo-
rate knowledge base consisting of data composed mainly
by its community members. This graph obeys the power
law. It contains 83,409,054 nodes and 293,351,870 edges.
We use the type of each subject as an element of our label
collection. This graph has totally 16,524 labels. The loading
time of freebase is 34,464ms.

The Impact on Query Length To test the impact of query
length, we vary the length from 5 to 25 and the results
are shown in Figure 7(a). From the results, the query time
increases with the increase of the length of the regular
expression approximately. The instable cases are caused by
the variety in the structure of randomly generated queries.
In the case of the same length of the regular expression,
different regular expression query has different execution
time. The reasons are as follows. On the one hand, longer
regular expression causes more logical operators, and more
physical operators will be generated. As the result, more
intermediate result sets will be loaded in the memory, and
more join operations are performed.

On the other hand, each regular expression is different
from other regular expressions with the same length, the
labels and the wildcards are not the same in each regular
expressions. Therefore, in the case of the same length of
the regular expression, the execution time are not the same.

2. https://www.graphengine.io/
To test the impact of query processing. This approach is only suitable for RPQs with more number of machines.

### The Impact of Graph Density
To test the impact of graph density, we vary the average degree from 4 to 64. The results are shown in Figure 7(d). From the results, the response time increases linearly with the average degree. This is caused by the increases in the size of intermediate results to handle. That is, with the increase of the average degree, the interconnections within the nodes increase and as the result, the number of intermediate results increases.

### The Impact of Label Number
To test the impact of label number, we choose 100%, 50%, 25%, 12.50%, 6.23% labels from the whole label set to generate label sets with 5163, 2581, 1290, 645, 322 labels. The experimental results are shown in Figure 7(e). From the results, the running time decreases significantly from the label number. The reason is that when label number increases, the number of nodes with the same label gets small and the number of nodes to be handled by the query decreases correspondingly.

### Speedup
To test the speedup of our system, we vary the machine number from 1 to 32 and the experimental results are shown in Figure 7(f). From the results, when the machine number increases from 1 to 8, the running time decreases significantly. While when the machine number gets larger, the running time increases slightly. This is because when the machine number is small, the time of cluster maintenance is relative small. Thus the acceleration effect of increasing machine number is significant. As a comparison, when the machine number gets large, the load of cluster maintenance is heavy and in such case, the running time gets slow. This shows that our system could achieve high performance with more number of machines.

### 7 Related Work
Since regular expression is a powerful form for query requirement description on sequences, structural queries with regular expression on data with complex structures, such as XML data and graphs, have been studied.

Some work focuses on the expressive power of query language. [2] surveys various syntax and semantics regular expression queries on graphs. nSPARQL [13] embed regular expression into SPARQL. Different from these work, our concern is the efficiency and scalability. [7] studied the difficulty of the evaluation, containment and subsumption with SparQL with embedded regular expressions.

Due to the importance, the regular expression path query (RPQ for brief) processing on large graphs has been studied. Some algorithms have been proposed to process RPQs efficiently. [5] uses regular expression to describe the relationship between the vertices in the graph pattern. It allows path queries with regular expressions formed with edge-labels. An index with size $O(|V|^2)$ is used for query processing, which is not suitable for large graphs. Additionally, the regular expression in [6] considers neither recursion of expressions nor ‘?’. [11] processes RPQ queries by traversal and uses rare labels to optimize the RPQ query processing. [8] studies the processing of RPQ based reachability queries, which contain Kleene closure. It processes queries by translating the query graph into relational operators including scans, projections and joins. The label-based index for reachability queries is also proposed to accelerate query processing. This approach is only suitable for RPQs...
with closure on single label instead of a clause. [17] processes RPQ by translating RPQs into recursion SQL queries. Waveguide [18] builds a cost-based optimizer for SPARQL queries with regular expression. It generate a query plan as the combination of finite automatas for a RPQ. The RPQ is processed by graph traversal guided by the query plan. [15] the approximate matching and relaxation of conjunctive regular path queries by introducing two new operators for flexible query processing.

All these algorithms aim to process RPQ on single machine without parallel paradigm. The scalability is limited. Additionally, the expressive power of the languages used by some approaches is limited. Different from these work, our system adopts parallel mechanism to process queries in full regular expression syntax on billion-node graphs.

Horton+ [14] adopts a parallel platform to process RPQ on graphs. It also decomposes the query into segments and joins their results.

8 Conclusion

In this paper, we study the problem of regular expression (RE) matching on large graphs, which is to retrieve pairs of vertices in the graph with the labels in the path between them satisfying the constraint of the RE in the query. We propose the methods to process the RE query on large graphs with the index sublinear to the graph size. To obtain the efficient query plan for regular expression query processing, we design the query optimization strategy based on the cost model in absence of the global structural statistical information for the graph. To process the RE queries on web-scale graphs, we also develop the parallel processing algorithm independency to the distribution of data with two simple network primitives. Experimental results demonstrate that our system can scale to billion-node graphs.

As the future work, we plan to study two problems. One is the efficient processing methods with the regular expression with wildcard, especially with the wildcard in the closure. The other is to study the efficient matching algorithm for graph patterns with regular expressions embedded in them.

References

[1] Facebook: One social graph to rule them all? CBS News. July 11, 2010.
[2] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. ACM Comput. Surv., 50(5):60:1–60:40, 2017.
[3] D. Chakrabarti and C. Faloutsos. Graph Mining: Laws, Tools, and Case Studies. Synthesis Lectures on Data Mining and Knowledge Discovery. Morgan & Claypool Publishers, 2012.
[4] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In SDM, pages 442–446, 2004.
[5] P. E. C. Compeau, P. A. Pezvner, and G. Tesler. How to apply de bruijn graphs to genome assembly. Nature Biotechnology, 29.
[6] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In ICDE, pages 39–50, 2011.
[7] G. H. L. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. In EDBT, 2016.
[8] A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling kleene: fast property paths in RDF-3X. In First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013, page 14, 2013.
[9] H. Hosoya and B. C. Pierce. Regular expression pattern matching for xml. In POPL, pages 67–80, 2001.
[10] H. Hosoya and B. C. Pierce. Xduce: A statically typed xml processing language. ACM Trans. Internet Techn., 3(2):117–148, 2003.
[11] A. Koschnieder and U. Leser. Regular path queries on large graphs. In Scientific and Statistical Database Management - 24th International Conference, SSDBM 2012, Chania, Crete, Greece, June 25-27, 2012. Proceedings, pages 177–194, 2012.
[12] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In VLDB, pages 361–370, 2001.
[13] J. Pérez, M. Arenas, and C. Gutierrez. rspanql: A navigational language for RDF. J. Web Sem., 8(4):255–270, 2010.
[14] M. Sarwat, S. Elnikeyti, Y. He, and M. F. Mokbel. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. PVldb, 6(4):1918–1929, 2013.
[15] P. Selmer, A. Poulovassilis, and P. T. Wood. Implementing flexible operators for regular path queries. In Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015., pages 149–156, 2015.
[16] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. PVldb, 5(9):788–799, 2012.
[17] N. Yakovets, P. Godfrey, and J. Gryz. Evaluation of SPARQL property paths via recursive SQL. In Proceedings of the 7th Alberto Mendelzon International Workshop on Foundations of Data Management, Puebla/Cholula, Mexico, May 21-23, 2013., 2013.
[18] N. Yakovets, P. Godfrey, and J. Gryz. WAVEGUIDE: evaluating SPARQL property path queries. In EDBT, 2015.
[19] N. Yakovets, P. Godfrey, and J. Gryz. Query planning for evaluating SPARQL property paths. In SIGMOD, 2016.
[20] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. PVldb, 6(4):265–276, 2013.
[21] Q. Zhang, W. Zheng, and M. R. Lyu. Flow-augmented call graph: A new foundation for taming API complexity. In FASE, pages 386–400, 2011.