GLog: A High Level Graph Analysis System Using MapReduce

Jun Gao*†, Jiashuai Zhou*†, Chang Zhou*† and Jeffrey Xu Yu‡§
*Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China
†School of Electronics Engineering and Computer Science, Peking University, China
‡Department of systems engineering & Engineering Management, Chinese University of Hong Kong
§Key Laboratory of High Confidence Software Technologies (CUHK Sub-Lab), Ministry of Education, China
Email: {gaojun,zjiash,zhouchang}@pku.edu.cn, yu@se.cuhk.edu.hk

***Abstract*—With the rapid growth of graphs in different applications, it is inevitable to leverage existing distributed data processing frameworks in managing large graphs. Although these frameworks ease the developing cost, it is still cumbersome and error-prone for developers to implement complex graph analysis tasks in distributed environments. Additionally, developers have to learn the details of these frameworks quite well, which is a key to improve the performance of distributed jobs.**

This paper introduces a high level query language called GLog and proposes its evaluation method to overcome these limitations. Specifically, we first design a RG (Relational-Graph) data model to mix relational data and graph data, and extend Datalog to GLog on RG tables to support various graph analysis tasks. Second, we define operations on RG tables, and show translation templates to convert a GLog query into a sequence of MapReduce jobs. Third, we propose two strategies, namely rule merging and iteration rewriting, to optimize the translated jobs. The final experiments show that GLog can not only express various graph analysis tasks in a more succinct way, but also achieve a better performance for most of the graph analysis tasks than Pig, another high level dataflow system.

I. Introduction

Large graphs emerge in different applications like social network, transportation network, and pose serious difficulty to the graph data management. These graphs have a large number of vertices (*e.g.,* more than 100 million vertices), and complex relationships among the vertices. On these large graphs, even the basic operations, such as triangle discovery [23], shortest path index construction [19], incur massive intermediate results and high computational cost, which can easily exceed the capability of a single computer.

The utilization of distributed data frameworks is inevitable in managing large graphs. The distributed frameworks, such as the MapReduce framework [14] or Pregel style systems [10], [25], offer functionalities like data storage, message transfer, job schedule, fail takeover, to developers. With the aid of these distributed processing frameworks, the developers need not build their distributed programs from scratch, and then the development productivity can be improved.

Although these frameworks ease the burden of developers in implementing graph operations, they still impose strict requirements on developers. Take Hadoop [2], an open-source implementation of MapReduce framework, as an example. Developers need to rewrite their methods using map and reduce functions with an advanced programming language, compile the codes and submit them to Hadoop. Since most of graph operations require sophisticated optimization strategies, it is not surprised that the corresponding MapReduce jobs are complex. When exceptions occur at the running time, it will be cumbersome and time-consuming to debug these distributed programs, at least in the current version of Hadoop [2]. In addition, it's very likely for developers to write inefficient MapReduce programs if they do not understand the internal data flow in Hadoop clearly.

A high level graph analysis language has potential to address these issues. The language allows developers to focus on the data flow instead of low level interfaces. At the same time, the query engine can optimize the evaluation plan according to the features of the framework, data distributions and clues provided by the language. Actually, many high level query systems have been developed. The representative ones include the SQL style query systems like Pig [8], [4], Hive [5], YSmart [21], and the declarative machine learning system like SystemML [26]. These high level query systems are attractive not only to the data analysts who have limited knowledge in coding low level languages, but also to developers. Take Yahoo as an example, roughly 60% of ad-hoc Hadoop jobs are submitted via Pig [4], a high level data flow system.

However, it is challenging to develop a high level graph analysis system in distributed environments. One fundamental issue is the lack of a well-accepted graph analysis language. The language should have a sufficient expressive power, as graphs are complex and their operations are flexible. At the same time, developers need a concise language. The existing graph analysis languages, such as graph algebra [11], graph pattern query [13], [15], PigLatin [8], [4], etc, cannot achieve both conciseness and expressive power ideally. For example, the operators in graph algebra [11] are described by nested expressions, which are not easy for users. PigLatin has a limited support to graph operations with iterations. In some cases, developers have to incorporate PigLatin into an external programming language to handle complex graph operations.

Another challenge is the evaluation performance of a high level graph analysis language. For the same graph analysis requirement, users may have different expressions. For the same expression, there are also various implementations in different distributed frameworks. In addition, graph operations always incur iterative jobs, while the number of jobs is a key factor to the performance in distributed frameworks [9], [6].

MapReduce framework is selected to study the high level graph analysis language in this paper due to two reasons. First, MapReduce framework has been the de facto standard in processing big data, and many companies have built their own Hadoop clusters [2]. It is convenient for end users to manage graphs using MapReduce framework. Second, MapReduce framework can handle massive intermediate results in graph analysis, which will pose a high pressure to the memory based systems [10], [25]. At the same time, it is not nature to extend these memory based frameworks to spill the intermediate data into disk. For example, the latest version of GraphLab [25] cannot support the swap disk mechanism when the memory is full. Notice that our work focuses on a framework-independent graph analysis language. The language can be evaluated in different distributed frameworks besides MapReduce.

In this paper, we propose a high level graph analysis language, named GLog, and design its evaluation method in Hadoop platform. Specifically, this paper makes the following contributions:

- We propose RG tables to mix the relational data and graph data, and design a high level analysis language, called GLog, on RG tables. GLog keeps the simple expression as the classic Datalog, and makes extensions on nested data access, explicit dataflow control and more built-in functions. (Section II)
- We devise a method to translate a GLog query into a sequence of MapReduce jobs. We formally define operations on RG tables behind GLog rules, and provide translation templates for these operations in MapReduce framework. (Section III)
- We design optimization strategies on the translated MapReduce programs. As the total number of MapReduce jobs impacts the performance seriously, we reduce the translated jobs by rule merging and iteration rewriting. (Section IV)
- We conduct various graph analysis tasks with GLog over both real-life and synthetic billion-edge graphs in a 28-node Hadoop cluster. The results show that GLog is a succinct language. Additionally, with the optimization strategies, GLog outperforms Pig on most of graph analysis tasks. (Section V)

The remainder of this paper is organized as follows: In Section II, we show RG tables and GLog language. In Section III and Section IV, we propose basic and optimized methods to translate a GLog query into MapReduce jobs respectively. In Section V, we conduct extensive experimental studies on real-life and synthetic data. Section VI reviews related work. Section VII concludes the paper and foresees the future work.

II. GLog: A High Level Graph Analysis System

In this section, we first show the architecture of a graph analysis system, discuss design choices, and propose RG table as the base for graph operations. Then we illustrate the functionalities of GLog query by a case, and present its high level syntax finally.

A. Architecture of High Level Graph Analysis System

We illustrate the architecture of our high level graph analysis system using MapReduce framework in Fig.1. In such a system, developers can express the data flow for a graph analysis task via a query q, and send q to the graph analysis engine. The engine parses q, translates q into Java codes for MapReduce jobs, and optimizes the codes within MapReduce framework. The translated query-specific codes are combined with common utilities to produce the final programs, which include a chain of MapReduce jobs and a job control program. The query engine will compile the programs, submit them to Hadoop, and send the results back to the users automatically.

Although the architecture is similar to other existing systems (*e.g.,* Hive [5], Pig [8], YSmart [21]), GLog has its own features in several ways. First, the underlying data storage is specific to graph data. We develop RG tables to mix relational data and graph data in a uniform way. Second, inspired by the well-known Datalog query [24], we design a language GLog for graph analysis. At the same time, we introduce explicit iteration statements into GLog, which make GLog more suitable in expressing flexible graph operations. Third, we develop strategies, such as iteration aware merging rules, to optimize the translated MapReduce jobs. These features will be discussed in detail in the following sections.

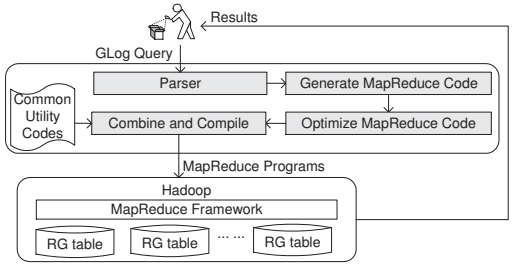


Fig. 1. Architecture of High Level Graph Analysis System

B. Design Choices

A series of issues need to be settled in designing a high level graph analysis system. The first one is the representation of a graph in distributed frameworks. A graph can be represented by a concise relational edge table, which records the IDs of ending vertices of all edges. The traditional Datalog can be easily expressed on the edge table [24]. However, the basic operations, such as the location of all incident edges for a specific vertex, still need a scan of the edge table, which is costly in distributed frameworks. On the other hand, a graph can be represented by a nested structure, which is taken by Pig [8], Socialite [16], graph algebra [11]. Although the nested representation enables more operations during a scan of one vertex, it will incur the complexity in both query expression and evaluation. In this paper, we attempt to balance the simplicity and expressiveness in the graph representation.

The second issue is whether we provide fixed graph operation APIs (*e.g.,* shortest paths, minimal spanning tree, Pagerank), or primitive graph operations (*e.g.,* selection, join, projection), to developers. Although the graph APIs can encapsulate implementation details, it is not flexible as new requirements, which are out of the scope of existing APIs, may emerge. In addition, users cannot develop their own methods

which are more suitable on their data set. Thus, in this paper, we will provide developers with the primitive graph operations via the extended Datalog rules [24].

Third, will recursion or iteration be introduced into the language to handle complex graph analysis tasks? In classic Datalog, these tasks are expressed by recursion, and the termination of recursions is determined by a fixed point strategy implicitly [24]. Although the recursion expression is clean and simple, it is not easy to encode sophisticated optimization strategies in graph operations using Datalog. For example, it is difficult to express Dijkstra's algorithm in finding the shortest path between two given vertices with recursion. In contrast, the iteration statement can offer a flexible termination control and provide more clues to the following query evaluation. In this paper, we plan to support the iteration statement with explicit termination condition in the language.

C. RG Table

We propose RG tables to balance the simplicity and expressiveness in the graph representation. The table form is kept due to its simple structure. The nested graph representation is also allowed as it can carry sufficient information, which is preferred by distributed frameworks. However, we forbid arbitrary nesting structure as it will create difficulty in the query expression and evaluation.

Specifically, we introduce a new nested data domain, graph vertex, into relational tables besides other atomic data domains. An instance of a graph vertex records incident edges of a vertex. In addition, the application related attributes are allowed to be annotated on the vertex, such as the distance to the source vertex in the shortest path discovery, or the parent vertex in the spanning tree construction. We can observe that the similar idea is adopted by the relational XML management system like SystemRX [17], where XML document is a new domain in the relational DBMS engine. Formally, the vertex domain can be defined as follows.

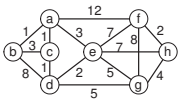
Definition 1: **Attribute Extensible Vertex.** An attribute extensible vertex takes the form of $u = \{A, E\}$. A is a set of attributes on u, and $id \in A$ is a default attribute for $u's$ identifier. E is an edge set $\{(v, c)\}$ for u, each of which is for an edge from u to v with its cost c. The projection results on v in E are referred to as $u.ToIDs$.

The edge $(v, c) \in E$ is not allowed to be further nested. The set E maintains the relationships from u to other vertices. For example, E can record incident edges for an original graph vertex, or can be empty for an intermediate vertex when edge information is not used in graph operations. In the following, we say that two vertices have the same domain if they have the same extensible attributes A, and two vertices are equivalent if they have the same values on the corresponding attributes. With attribute extensible vertices, RG table can be defined as follows:

Definition 2: **RG Table.** A RG (Relational-Graph) table takes the form of $R = (A_0, \ldots, A_n)$. Each attribute $A_i (0 \leq i \leq n)$ is either with the attribute extensible vertex domain (shorten as vertex domain), or with an atomic data domain. An attribute $A_i (0 \leq i \leq n)$ is called a flat attribute if its data domain is atomic. The attributes inside the vertex are called nested attributes. Both flat attributes and nested attributes are table R's attributes.

RG table is the underlying data model of GLog system. Traditional relational tables are special cases of RG tables. For tuples r_1 and r_2 in a RG table, r_1 and r_2 are equivalent if their corresponding components are equivalent. RG table has bag semantics. That is, multiple equivalent tuples are allowed in one RG table.

We illustrate RG tables in Fig.2. Fig.2(b) shows the RG table for a graph in Fig.2(a). Each tuple in the table contains an attribute extensible vertex u, which records incident edges of u. Fig.2(c) shows a RG table used in the landmark index construction [19], where *Landmark* is a landmark vertex ID with an atomic domain, *VisitedVertex* records a visited vertex with an attribute extensible vertex domain (its edge set E is empty). For example, the tuple $(b, (c, 3, b, 0))$ represents a landmark vertex b and a visited vertex c. The values $(3, b, 0)$ are for extensible attributes on vertex c, including the distance from b to c, the predecessor of c, and the expansion status (0 for not-expanded) respectively.



(a) Original graph

Vertex(id,{(toid,cost)})
(a,{(b,1),(c,1),(e,3),(f,12)})
(b,{(a,1),(c,3),(d,8)})
... .

(b) RG table for graph

Landmark	VisitedVertex(id,cost,pre,sign)
b	(c,3,b,0)
a	(c,1,a,0)
...	...

(c) RG table for visited vertex

Fig. 2. RG Table Examples

D. Case Study: Optimized Landmark Index Construction

Roughly speaking, a GLog query on RG tables is a combination of explicit iteration control statements and Datalog rules for graph operations. Before giving detailed descriptions on GLog, we show the expressive power of GLog via a query for the landmark index construction intuitively.

The landmark index can be used to estimate the shortest distance between two given vertices efficiently [19]. An index is a collection of landmark vertices, each of which records the shortest distances to all other vertices. With the landmark index, the distance between any two vertices in a graph can be estimated by triangle inequality. The landmark index construction is suitable in MapReduce framework, as a large number of vertices need to be visited during the construction of the landmark index. Let l be the number of landmark vertices, n be the number of vertices in the graph. The index construction requires $O(ln)$ intermediate results.

The landmark index construction using GLog is shown in List 1. Suppose that the IDs of landmark vertices have been stored in a RG table *Source*. In rule 1, a RG table *All* (s, v_1) is initialized, where s is the ID of a landmark vertex, v_1 is a vertex with its ID s. The extensible attributes of v_1 including $cost$, pre, and $sign$ are also initialized, which respectively stand for the distance to the landmark vertex, the predecessor to the landmark vertex, and the sign for whether the vertex has been expanded or not. The sub-part after "|" includes update operations on *All* (s, v_1).

We initialize a variable *num* in the 2-nd rule. From this line, we see that GLog supports control variables, which can be used in the following GLog rules or the termination condition.

With such a feature, GLog provides more flexible data flow control than the classic Datalog [24].

The optimization strategies are highly needed for the landmark index construction in MapReduce framework. As the total number of MapReduce jobs makes a big impact on the performance [9], [6], we prefer BFS(breadth-first-search) instead of the classic Dijkstra's algorithm in the shortest path discovery. At the same time, only partial vertices are selected as the frontier vertices for the next expansion to avoid a huge search space in BFS. In addition, the expansion in parallel may produce multiple versions for the same vertex. We require a mechanism to remove unnecessary versions so that the following search space is reduced.

Listing 1. Landmark Index Construction

```
1:All(s, v1):−Source(s)
    |v1.id = s , v1.cost = 0 , v1.pre = s , v1.sign = 0;
2:num:−1;
3:Do {
4: Frontier(s, v0):− All(s, v0) , v0.sign = 0 ,
    v0.cost ≤ num ∗ w | v0.sign = 1;
5: New(s, v2):−Frontier(s, v0) , graph(v1) , (y, cost2) ∈ v1.E ,
    v0.id = v1.id | v2.cost = v0.cost + cost2 , v2.id = y ,
    v2.pre = v0.id , v2.sign = 0;
6: All(s, v0):−New(s, v0) or All(s, v0) or Frontier(s, v0);
7: All(s, v0):−PSA(All(s, v0) , partition by s, v0.id ,
    sort on v0.cost asc, on v0.sign desc, rank() = 1);
8: num:−num+1;
9: Remain(v0):−All(s, v0) , v0.sign = 0
10:}While (Exists(Remain(v0)))
11:Return All(s, v0)
```

These optimization strategies can be expressed by GLog. We see the rules from line 3 to line 10 with an explicit "Do-While" statement. We select frontier vertices from all visited vertices in line 3, when the distances on vertices are no more than *num**w. Here, w is a constant assigned by users, which can be set to the average weight of all edges. The signs for the frontier vertices are also changed in line 4, which indicates that the frontier vertices have been expanded. We make one expansion from frontier vertices along their edges in the graph to find newly visited vertices in the 5-th rule. The newly visited vertices, the frontier vertices, and the existing visited vertices are combined into *All*. Since a vertex can be visited by different paths, there may be multiple versions for the same vertex. *PSA* function is used to select the versions with the minimal cost, and keep one with $sign$=1 if possible. The iterations terminate when there are no newly visited vertices ($sign$=0). At that time, *All* keeps all visited vertices with their shortest distances to all landmark vertices. Note that it is difficult to express these optimizations using the classic Datalog.

E. Overview of GLog

Now, we describe overall functionalities of our graph language GLog. Due to the limitation of space, only the partial high-level syntax is illustrated in List 2. A GLog query consists of a sequence of rules, including RG rules (RGRule), variable rules (VarRule) and control rules (DoWhile, ReturnRule). Each RG rule accepts one or multiple RG tables, and yields another RG table. Like a Datalog rule, a RG rule has a head and body. The body contains its query part and update part, separated by "|". The query part consists of a sequence of predicates, each of which takes the form of RG table, variables for nested attributes of vertices, the relationship between variables, or built-in functions like *PSA*. The predicates on variable, along with the built-in functions including *Exists* and *ExistsNew*, can be used as termination conditions in iterations. The variable rules and control rules are similar to their counterparts in any programming language. Since RG rules play an important role in a GLog query, they are also referred to as GLog rules directly in the following.

Listing 2. Partial High level syntax for GLog

```
1:GLog= (RGRule|VarRule|DoWhile)∗,ReturnRule;
2:RGRule= Head:−Body
3:DoWhile= Do (RGRule| VarRule)∗
          While (Termination())
4:Head= RGTable
5:Body= QueryPart (| UpdatePart)?
6:queryPart=(RGTable|VarPredicate|EdgeIn|PSA)∗
7:RGTable= TableName(col+)
8:EdgeIn= (toid, cost) ∈ vertex.E
9:VarRule=variable:−expression
10:PSA= PSA(RGTable,(partition by col∗), (sort
   by col∗),(VarPredicate|rank(col))|(aggregation∗)
11:Termination= ExistsNew(RGTable)|
             Exists(RGTable)|VarPredicate
12:ReturnRule = Return(RGTable)
13: ... ...
```

Actually GLog is an extension to the classic Datalog. We now summarize major differences between them. First, GLog uses iteration instead of recursion to handle complex and flexible graph operations. *Do-while* statement can control the iterations in graph operations in an explicit way. Variable rules enable value assignment to variables, which can be used in GLog rules or termination condition of iterations. With the introduction of these features, the original evaluation of recursive Datalog rule will be changed. That is, for each rule r in a Do-while iteration block, r is invoked only once in each iteration, even if the table in the head of r also appears in the body of r.

Second, GLog rules run on RG tables instead of on relational tables, and then the access to the nested attributes is needed in GLog rules. For a vertex u in a RG table, its nested attributes can be accessed using the expressions like $u.id$ or $(tid, cost) \in u.E$. These variables can be further referred in the same GLog rule.

Third, more built-in functions are introduced into GLog. *Exists(R)* indicates whether a RG table R has elements or not. *ExistsNew(R)* detects whether a RG table R has new elements compared with R's previous version. Both functions can be used in the termination detection of iterations. Another important function, *PSA*, is an extension to the classic group-by and aggregation function. *PSA* function can partition the tuples into different groups, make selections among sorted tuples on specified attributes, or perform aggregation functions like MIN and SUM, in each group.

GLog is more expressive than the classic Datalog. A non-recursive Datalog rule can be expressed in a similar way in GLog. For any recursive Datalog rule, we can rewrite it using a *Do-while* statement, and use *ExistsNew* or *Exists* to detect the termination of iterations. In contrast, a GLog query with iterations may not be expressed by classic Datalog rules. For example, without the support from external functions, the classic Datalog cannot express a dynamic predicate related with the number of iterations, such as $v_0.cost \leq num * w$ in the rule 4 in List.1.

III. TRANSLATION OF GLOG QUERY

In this section, we give an overview of the translation method for a GLog query, then focus on the translation templates for different GLog rules, and finally present the complete translation approach.

A. Overview of Translation

We first need to assign a key k for each tuple t in a RG table R, as t takes the form of key-value pair $k \rightarrow r$ in a HDFS file. k can be the vertex ID when R is for the original graph, or is determined in the user-defined mapper or reducer when R is for the intermediate result of a MapReduce job.

The translation from a GLog query Q to its MapReduce jobs can be divided into two phases, namely static analysis and code generation. In the first phase, we extract all different RG tables and their attributes from Q. The domains of attributes are inferred from operations on attributes. Take the 1-st rule in List 1 as an example. We will find a RG table named *All*. It has two attributes, one with an atomic domain and another with a graph vertex domain. The nested attributes of the vertex, like $cost$, pre and $sign$, can be obtained from the body of the RG rule. At the same time, we know the data domain of pre is String, and the domains of other attributes are Integer from the operations in the body.

The second phase is to translate Q into MapReduce codes, which include a job control program P along with multiple MapReduce jobs. For a variable v in a variable rule, we can translate v into a context variable [2] in P, as the context variable can be accessed by mappers or reducers at the running time. The *Do-while* rule and *Return* rule are directly translated into the counterparts in the job control program P. For a GLog rule, we can translate it into one or multiple MapReduce jobs with the templates discussed below.

The translated codes in the second phase will be combined with common utilities to yield the final MapReduce program. The common utilities include the codes used for all queries, which can be designed in advance. The major parts of common utilities are related with RG tables, such as the domain definition of attribute extensible vertices and RG tables, their serialization and de-serialization methods, etc. In addition, the management of data directories for RG tables is included in common utilities. We need to keep two latest data directories for a RG table R in iterations, if an *ExistsNew(R)* function appears in the termination rule.

B. Translation Templates

We can extend relational algebra to support the evaluation of a GLog rule. A classic Datalog rule can be translated into relational algebra expression [24], and similarly, the evaluation of a GLog rule can be supported by the operations like selection, projection, join, union, aggregation, etc, on RG tables. There are many studies on the translation of relational operators to MapReduce jobs [5], [8], which can be borrowed in this paper. The issues introduced in GLog include the nested structure of RG tables and new built-in functions, like *ExistsNew* and *PSA*.

As a RG table may take a nested structure, and the nested attributes are allowed to be accessed in GLog rules, we first show two operations which make transformations between a RG table and a relational table.

Definition 3: **Unnest and Nest.** Let c_0 be any component of a tuple t in a RG table R. c_0's unnested form, $\mathcal{U}(c_0)$, is $\{(c_0.a_0, \ldots, c_0.a_k,\ c_0.toid, c_0.c)\}$ if $c_0 = \{A, E\}$ is with a vertex domain, where $a_0, \ldots, a_k \in A$, $(toid, c) \in E$. Otherwise, $\mathcal{U}(c_0)$ is c_0. t's unnested form, $\mathcal{U}(t)$, is the Cartesian production of unnested forms of all t's components. R's unnested form, $\mathcal{U}(R)$, is the union of unnested forms of all tuples in R. The nest operation $\mathcal{N}(\mathcal{U}(R))$ is an inverse operation to unnest operation.

The unnest and nest operations offer a way to precisely define the semantics of basic operations like selection, projection, etc, on RG tables. For any operation on RG tables, its results can be obtained by converting RG tables into relational tables with a unnest operation, evaluating corresponding relational operation on the relational tables [12], and converting the results back to a RG table with a nest operation.

Definition 4: **Selection.** Let R and S be RG tables, F be the predicates on the attributes in R. The selection result $S = sel_F(R)$ contains $\{\mathcal{N}(t) | t \in \mathcal{U}(R), F(t) = true\}$.

Let r be a GLog rule with a selection operation on RG table R. r is translated as follows. We locate the latest data directory of R and use it as the input directory for a mapper. For any input key-value pair $k \rightarrow v$, we obtain the meta data collected in the static analysis phase, convert v to a tuple $t \in \mathcal{U}(R)$, and apply the predicates in F on t. When a tuple t meets all the predicates, $k \rightarrow v$ is emitted from the mapper. After an identity reducer, the tuples are kept to a directory for the RG table S. We can design another translation template for the selection operation, which uses an identity mapper and implements the selection in the reduce side. Note that we need not apply the unnest and nest operations explicitly in the implementation of selection, as the programming language can access nested attributes easily.

Definition 5: **Projection.** Let R and S be RG tables. The projection result $S = prj_{F_0(A),\ldots,F_n(A)}(R)$ contains $\{\mathcal{N}(t[F_0(A), \ldots, F_n(A)]) | t \in \mathcal{U}(R)\}$, where $F_i(A) (0 \leq i \leq n)$ is constant or an expression on attributes in R.

The projection operation on RG tables has several features. First, the update part in the body of a GLog rule is supported by a projection operation. That is, we translate the update expressions into the projection expressions. Second, the projection enables the conversion between nested attributes and flat attributes in a table. We can assign a value to a nested attribute from a flat one, and vise versa, in the update part.

The translation of a projection can be described as follows. We locate the meta data for RG table R and S. For each key-value pair $k \rightarrow v$ in a mapper, we convert v into a tuple t_r in R. According to the projection rule, we extract the required attributes from t_r, evaluate the expressions, and generate another tuple t_s for S. The key-value pair $k \rightarrow t_s$ is emitted from the mapper. Similar to the selection operation, the projection can also be done in the reduce side.

Definition 6: **Join.** Let R and S be RG tables. The join result $Join_{R[A],S[B]}(R, S)$ contains $\{\mathcal{N}(rs) | r \in \mathcal{U}(R), s \in \mathcal{U}(S), r[A] = s[B]\}$, where A and B are table attributes.

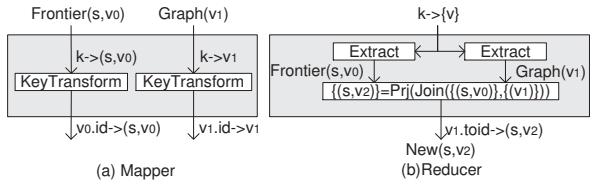


Fig. 3. MapReduce Implementation for Join

We illustrate a translation template for the join operation using the 5-th rule in List 1 in Fig.3. In the mapper, we locate the meta data for RG table *Frontier*, *Graph* and *New*. The data directories of *Frontier* and *Graph* are used as the input directories for mappers. We transform the input element $k \rightarrow (s, v_0)$ into $v_0.id \rightarrow (s, v_0)$ for table *Frontier*, where $v_0.id$ is the join attribute. We do a similar key transformation on tuples in table *Graph*. These transformed elements are emitted from mappers, and shuffled into reducers according to their keys. In the reducer, the input element takes the form of $k \rightarrow \{v\}$. We extract tuples $v_0 \in$ *Frontier* and $v_1 \in$ *Graph* from $\{v\}$, compose a vertex v_2 for *New* from v_0 and v_1, and output the element $v_2.id \rightarrow (s, v_2)$ in the reducer. Such a join is known as a repartition join [22]. Due to the limitation of space, we omit the parameters of operations in the figure.

The join operation can be also translated in other ways. For example, we can use a broadcast method to evaluate the join operation, when one of the tables, such as R, is small enough [22]. In such a method, we can put R into a hash table. The join results can be generated by one scan of table S in the map side, in which the hash table can be accessed at the running time.

Definition 7: **Difference.** Let R and S be RG tables. The difference results $R - S$ are $\{\mathcal{N}(r))|r \in \mathcal{U}(R), r \notin \mathcal{U}(S)\}$.

The difference operation takes the bag semantics by default, as multiple equivalent versions are allowed in a RG table. We still use nest and unnest operations to define the difference operation in order to make it compatible with other operations. The difference operation is the base for the built-in function *ExistsNew*(R). That is, we store the results of R of two latest iterations into different directories, and perform the difference operation to compare these directories to check whether there are new elements found in the last iteration.

The difference operation can be translated as follows. We locate the meta data of S and R, and use the data directories of two tables as the input directories of mappers. We introduce a hash function h on the tuple values used in the mapper, as the difference operation needs to compare all components of two tuples. In the mapper, we scan each element $k \rightarrow v$, and emit $h(v) \rightarrow v$. For any two equivalent tuples, they will be sent to the same reducer. In the reducer, we extract tuples for S and R from the element $k \rightarrow \{v\}$ respectively, and implement the difference operation.

Definition 8: **PSA.** Let R and S be RG tables. *PSA*$_{P,S,A}(R)$ partitions the tuple set of $\mathcal{U}(R)$ into groups $G_0, \ldots, G_n$ based on the attributes P, selects tuples with the predicates S or performs aggregation functions A in $G_i (0 \leq i \leq n)$, and unions the results from all $G_i (0 \leq i \leq n)$ into S'. $S = \mathcal{N}(S')$ is the result of *PSA*$_{P,S,A}(R)$.

The *PSA* operation is an extension to the group-by and aggregation functions in relational algebra. As we have unnested the table R, the table partitioning has the same semantics as the group-by on the relational table. In each group, we can select partial tuples or perform aggregation functions. The selection can be based on the component values or the tuple position in a sorted tuple sequence. For example, a parallel graph search will produce multiple versions for the same vertex, and we can use *PSA* to partition the vertex set according to the vertex ID, sort versions according to the given criteria, and select the desired version in each group.

The *PSA* operation can be translated into the MapReduce jobs as follows. For each element $k \rightarrow v$ scanned in the mapper, it is transformed into another key-value pair $k_1 \rightarrow v$, where k_1 includes the partition attributes specified by the *PSA* operation. In the reducers, the elements with the same key are grouped together. We can evaluate selections or aggregation functions in each group, and yield the results of all groups.

There are other basic operations, such as set union and intersection, which have the similar data flow as the difference operation. The built-in function *Exists*(R) returns whether the table R is empty or not. It can be implemented by mappers or reducers which adjust the status of a pre-defined counter in Hadoop [2]. Due to the limited space, we omit the details of these translation templates.

C. Translation Approach for GLog

Based on the translation templates above, we give an entire translation approach for a GLog query in Algorithm 1. The translation method first locates all RG tables in a GLog query Q in line 1, and initializes a job control program P in line 2, whose data flow will be the same as that of Q.

Algorithm 1: Basic Translation for GLog Query

Input: A GLog query Q.
Output: MapReduce Programs.
1 Statically analyze Q to collect the meta data for all RG tables in Q;
2 Initialize an empty job control program P;
3 **for** *Each Rule* r *in* Q **do**
4 **if** r *is a variable rule* **then**
5 Add a context variable into P;
6 **if** r *is a Do-while rule* **then**
7 Add a Do-while statement into P;
8 **if** *Exists or ExistsNew is in the termination condition* **then**
9 Initialize a counter $c_r = 0$ in P;
10 Add a template MapReduce job for *Exists* or *ExistsNew* which adjusts c_r in reducers;
11 Replace the termination condition with a predicate on c_r in P;
12 **if** r *is a GLog rule* **then**
13 Determine operation type for r;
14 Generate corresponding MapReduce jobs J_r for r using templates;
15 Add a method to invoke J_r in P;
16 Return P and generated MapReduce jobs;

We translate GLog rules in Q one by one in the iterations from line 3 to line 15. When the rule is a variable rule, the translation is simple, as shown in line 4 and 5. When the rule is a control rule, and *Exists* or *ExistsNew* is used in the termination detection, we introduce a user-defined counter c_r, invoke a MapReduce job to change c_r according to the translation template of *Exists* or *ExistsNew*, and terminate iterations based on c_r. When the rule is a GLog rule, we detect the operation types behind the rule by analyzing all RG tables and the relationships among them in the body, and generate the corresponding MapReduce jobs using templates.

Take the landmark index construction in List 1 as an example. The program to control MapReduce jobs is illustrated in Fig.4. The solid directed lines between boxes represent the order of operations. The boxes filled with the grey color are for MapReduce jobs, with the input data in the left and output data in the right. Totally, we have 6 MapReduce jobs in the iterations, each of which is for one GLog rule or *Exists* function in the control rule. These separated MapReduce jobs are not efficient in distributed frameworks. We will discuss how to merge these generated jobs in the next section.

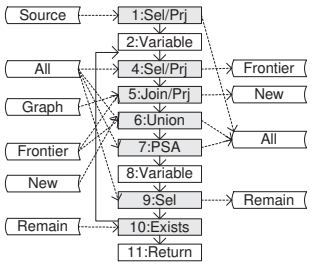


Fig. 4. Initial Plan for Landmark Indexing

IV. Optimized GLog Translation

In the section above, we can translate a given GLog query into MapReduce jobs. As the number of MapReduce jobs impacts the performance, we attempt to optimize the translated codes via rule merging and iteration rewriting. We will see that the 6 original iterative MapReduce jobs in Fig.4 can be merged into one with the strategies in this section.

A. Rule Merging

The reduction of MapReduce jobs is an important optimization strategy [6], [9]. As we know, each MapReduce job has its own overheads in scheduling, task set-up and clean-up. Besides, one table may be scanned multiple times by different MapReduce jobs, which leads to redundant computation cost [21]. When a job is iterative, these extra overheads are more considerable. Note that the job reduction strategy is also effective in other frameworks, such as Pregel [10], where the performance of distributed jobs is also related with the number of super-steps.

One way to reduce the total number of translated MapReduce jobs is to reduce GLog rules by merging, as each GLog rule is translated into at least one MapReduce job in the basic method. Due to interactions among the rules, we first find the scope of a GLog rule before merging. Take the 4-th rule in List 1 as an example. The table *Frontier* also appears in the body of the 5-th and 6-th rule.

Definition 9: **Dependent rule.** Let Q be a GLog query, r be a rule to derive a RG table T, a rule r_d depends on r if and only if T is in the body of r_d, and there is no other rule with T as its head between r and r_d.

Let r be a rule with T in its head, r_d be a dependent rule of r. We try to merge r and r_d into a new rule r'_d by replacing T in r_d with the body of r. We say the merging is worthwhile when $jobs(r'_d) < jobs(r_d)+jobs(r)$, where $jobs()$ returns the total number of translated jobs for a given rule. We classify operations on RG tables in order to distinguish which case the merging is worthwhile.

Definition 10: **1-side/2-side Operation.** Let op be an operation on RG tables. op is a 1-side operation, if op can be implemented by a transformation in a mapper followed by an identity reducer, or an identity mapper followed by a transformation in a reducer. Otherwise, op is a 2-side operation.

The selection, projection, union on RG tables are 1-side operations. Their transformations can be done in the map or reduce side. The other operations, like *PSA* and difference, are 2-side operations. The 1-side or 2-side property of an operation is also related with its implementation. For example, the join operation using a repartition method is 2-side, while the join operation with a broadcast method is a 1-side operation.

We can determine whether the merging is worthwhile according to the 1-side/2-side properties of operations. Let op_0 be an operation for a rule r_0, op_1 be an operation for a rule r_1, and r_1 be dependent on r_0. When either op_0 or op_1 is a 1-side operation, op_0 can be merged into op_1. For example, the transformation of op_0 can be done before that of op_1 in the mapper in the merged operation, if op_0 is a 1-side operation. However, when both operations are 2-side operations, we cannot merge op_0 and op_1 directly, as the merged operation cannot reduce the number of original MapReduce jobs.

Take the operations in the landmark index construction in Fig.4 as an example. The operations for the rules 4, 5, 6 can be merged into one operation op_m, implemented by one MapReduce job in Fig.5. The operations for the 4-th rule and 6-th rule are 1-side operations, while the 5-th rule needs a 2-side operation. The 4-th rule for table *Frontier* is merged into the 5-th and 6-th rule. And the 5-th rule for *New* is further merged into the 6-th rule. In the merged operation op_m, we observe that the transformation for the 4-th rule is done in the mapper, the transformation for the 5-th rule is done in both the mapper and reducer, and the transformation for the 6-th rule is done in the reducer after the transformations of other operations. The merged operation op_m is a 2-side operation, and we cannot merge op_m and another 2-side operation *PSA* in the 7-th rule. In fact, the newly visited vertices generated in join operation in the 5-th rule may have different IDs, and we cannot apply the *PSA* selection on them directly in the same reducer.

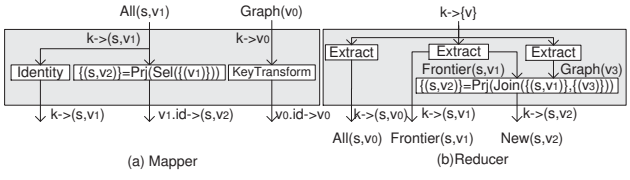


Fig. 5. Mapper and Reducer in Merged Operation

B. Iteration Rewriting

More operations are allowed to be merged if we rewrite the iterations. Still take the landmark index construction in List 1 as an example. We can put the rules 4, 5, 6 before the iteration, and start iterations from the 7-th rule followed by the rules 4, 5, 6. Now, a 2-side *PSA* operation for the 7-th rule is before a 2-side join operation in the iteration. We analyze the data flow between them. The *PSA* operation partitions the vertices in *All* according to $\{s, v_0.id\}$ into different groups, and emits the vertices with $\{s, v_0.id\}$ as the key in the mapper, where s is a landmark vertex ID and $v_0.id$ is a visited vertex ID. The join operation for the 5-th rule emits the vertices in *Frontier* (the results of a selection on *All*) with the vertex ID, $v_0.id$, as the key in the mapper. Then the shuffled results for the *PSA* in line 7 can be reused in the same reducer for the join operation in line 5, if we use $\{v_0.id\}$ as the partition attribute in the *PSA* operation, and defer partitioning on $\{s\}$ in the reducer. Thus we can use one MapReduce job to represent two operations. Formally, we give the conditions on which two 2-side operations can be merged.

Definition 11: **Shuffle-Shared Operations.** Let op_0 and op_1 be two 2-side operations, T_0 be an input table for op_0, T_1 be the output table for op_0 and an input table for op_1, k_0^m be the output key of T_0 in the op_0's mapper, k_0^r be the output key of T_1 in the op_0's reducer, k_1^m be the output key of T_1 in the op_1's mapper. op_0 and op_1 are shuffle-shared operations, if $k_0^m = k_0^r$ and $k_0^m \supseteq k_1^m$.

Suppose that we merge op_0 and op_1 into one MapReduce job op_2. We require $k_0^m = k_0^r$ and $k_0^m = k_1^m$, since we attempt to use T_0 instead of T_1 as the input table of op_2, and the transformations in the op_0's reducer and in the op_1's reducer can be done in the op_2's reducer. In order to enable more operations shuffle-shared, we relax the conditions by allowing $k_0^m \supseteq k_1^m$. In such a case, we partition T_0 using k_1^m in the op_2's mapper. In the op_2's reducer, we first extract tuples according to their sources. For the tuples from T_0, we further partition the tuples using the remaining attributes $k_0^m - k_1^m$, and perform the original transformation in op_0's reducer on the partitioned tuples, whose results further participate in the original transformations in the op_1's reducer.

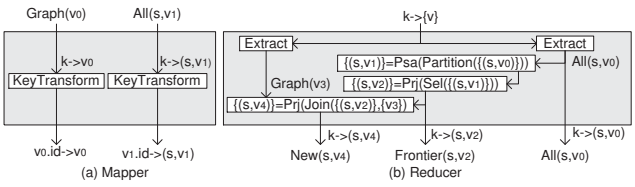


Fig. 6. Mapper and Reducer in Merged Operation

We illustrate a merged result in Fig.6, if we put the 7-th rule for a *PSA* operation before the rule 4, 5, 6 in Fig.4. We use the partition attributes $\{v_1.id\}$ instead of $\{s, v_1.id\}$ in the *PSA* operation, so that the shuffled results in the *PSA* operation can be shared by the join operation. The partition group is proper for the join but coarser for the *PSA* operation. Then in the reducer, we partition the tuples according to the remaining attributes $\{s\}$ for the *PSA* operation, whose results results will be transformed according to the rule 4, 5, and 6.

C. Optimized MapReduce Program Translation

Now, we give the complete algorithm for the optimized MapReduce program translation. We merge and remove the rules whose operations are 1-side in line 2. For any rule r with a 2-side operation, we detect whether the operations for r and any $r's$ dependent rules are shuffle-shared, and remove r after r is merged into its dependent rules if conditions are met from line 3 to line 7. If there still remain multiple operations in an iteration, these operations will be 2-side operations and any two neighboring operations cannot be shuffle-shared. We then detect whether the operations for the last rule and the first rule in the iteration can be shuffle-shared. If so, we rewrite the iteration, and merge these two rules from line 8 to line 10. Finally, we generate MapReduce jobs and a job control program using the basic method.

Algorithm 2: Optimized GLog Translation

Input: A GLog query Q.
Output: Optimized MapReduce Programs.
1 For each rule r in Q, label its 1-side/2-side property;
2 For any rule r with 1-side operation, remove r after r is merged into all its dependent rules;
3 **for** *each rule r with a 2-side operation* **do**
4 Locates $r's$ dependent rules R_d;
5 **for** *each rule $r_d \in R_d$* **do**
6 Detect whether the operations for r and r_d are shuffle-shared;
7 r is merged and removed, if the operations for r and any $r's$ dependent rules are shuffle-shared;
8 **if** *the operations of the last and the 1-st rule in an iteration are shuffle-shared* **then**
9 Rewrite the iteration to let the last rule be the 1-st rule;
10 Merge the 1-st and 2-nd rule in the iteration;
11 Invoke Algorithm 1 to produce MapReduce jobs J for all rules and a job control program F;

The final optimized plan for the initial one in Fig.4 is illustrated in Fig.7. We rewrite the iteration by putting the rules 4, 5, 6 before the iteration. The original 1-st rule along these rules can be implemented by one operation. The remaining rules in the iteration followed by the rules from line 4 to line 6 are implemented by another operation. The *Exists* function is avoided by introducing a counter into the job control program, which will be changed by the reducer in the translated MapReduce job for the original 9-th rule. Finally, we have only two MapReduce jobs for the landmark indexing, one before the iteration, and another in the iteration.

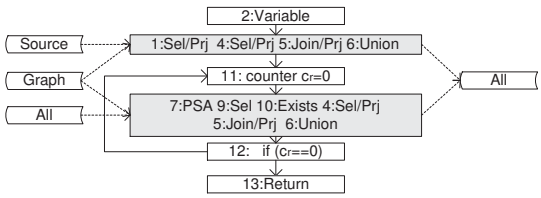


Fig. 7. Optimized Plan for Landmark Indexing

V. Experimental Evaluation

In this section, we first study the conciseness and expressiveness of GLog language, and then focus on the performance and scalability of GLog queries in Hadoop platform.

A. Experimental Setup

Environment. All experiments are evaluated on a 28-node cluster. Each node has two 2.60GHz AMD Opteron 4180 processors with 48GB of RAM and 10TB hard disk. We install SUSE Linux Enterprise Server 11 and Java 1.7 with a 64-bit server JVM on each node. All these nodes are connected to a 1GB network switch.

We use Hadoop version 0.20.3 and configure the cluster to run 6 map and 6 reduce slots concurrently on each node. Thus, we have at most 168 map tasks and 168 reduce tasks running concurrently in the cluster. The number of reducers for each job is set to 95% of the maximum slots, namely 160 reducers if not explicitly mentioned. The block size of HDFS is set to 64M and each block is replicated 3 times. We allocate 2G main memory to each mapper and reducer, and compress outputs of mappers and reducers.

Graph Operations. Besides the landmark index construction, we select other six representative graph analysis tasks, including *Reachability*, *Triangle*, *Mutual Neighbors*, *Connected Components*, *PageRank*, *Hub and Authority*, to evaluate GLog language. All operations are performed on directed graphs except triangle and connected components discovery, which are on undirected graphs. For landmark index construction (reachability query), we evaluate tests with randomly selected landmarks (source/target vertices) for many rounds, and report the average results.

Datasets. We use both real world graphs and synthetic datasets to test our approach. *livejournal* is a friendship network of an on-line community site LiveJournal. *twitter-2010* is a follower-following topology in the social network Twitter [18]. *uk-2007-05* is a time-aware graph which combines 12 monthly snapshots of the .uk domain collected by DELIS project [20]. *random* graph is generated as follows. Let n and m be the number of nodes and edges respectively, we randomly select the source and target node for m edges among n nodes. We assign a random weight in [1,100] to each edge in all above graphs. Some statistics about these graphs are summarized in Table I. We also report the graph transformation time from edge table representation to RG table representation in the last column. Such a transformation can be done by one MapReduce job.

TABLE I. Statistics of Graph Datasets

DataSet	# Nodes	# Edges	Storage(GB)	Time(Sec)
livejournal	4,847,571	68,993,773	0.73	91
twitter-2010	41,652,230	1,468,365,182	12.1	250
uk-2007-05	105,896,555	3,738,733,648	30.5	515
random	1.0×10^9	6.0×10^9	71.5	1167

Competitors. We compare GLog with handwritten MapReduce programs and another high level dataflow system Pig [3]. Pig is selected as it can process nested data, while other systems [5], [21] are on structured data. We select the similar data model like RG table as underlying storage structure in Pig, express the above 7 representative graph analysis routines, use the latest Pig 0.11.1 to compile these dataflow programs. As Pig has a weak support to the iterations, we embed Pig scripts in Java programming language. All GLog and Pig scripts run on the same Hadoop platform with the same configuration parameters.

We also evaluate these queries on Giraph 1.0 [1], an open source implementation of Pregel, to illustrate differences between the underlying frameworks. We find that Giraph cannot load large graphs, like *uk-2007-05* and *random*, into memory, if we set the maximal memory used by each worker in Giraph to 2G (the same value used by GLog). Then, the reported results below are collected on Giraph with 8G main memory for each of 50 workers.

In the following experiments, we first show the GLog query is quite succinct and has a sufficient expressive power. Second, we verify the effectiveness of optimization strategies and then compare the performance and scalability of GLog query engine on Hadoop with handwritten Java programs, Pig and Giraph.

B. Studies on Basic GLog Translation

Table II presents the number of lines of codes for GLog queries, the translated MapReduce jobs in Java, the handwritten Java programs, PigLatin scripts and Giraph implementation respectively. The common codes for RG table and additional utilities are not counted in the translated MapReduce jobs since they can be shared by all GLog queries once written. We can observe that all these graph operations can be easily expressed by 2 to 13 lines of GLog rules while the translated Java programs need many more lines. As for the carefully handwritten Java programs and Giraph scripts, they still take hundreds of lines of codes to implement the same queries. For example, a 2-line GLog query for mutual neighbors discovery will be translated into 186 lines of codes and we can reduce it to 150 lines by ourselves. With Giraph, we can further reduce them to 133 lines. As for Pig, we just count the lines of PigLatin scripts and exclude the external Java codes which control the data flow of the PigLatin scripts. The number of lines of PigLatin scripts is still more than that of GLog. Furthermore, we have to incorporate PigLatin scripts into external Java codes for iterative jobs, which also increases the burden on developers. All these results clearly show that GLog is a succinct language and can be used in various graph analysis tasks.

Table II also shows the query translation time of GLog in the last column. It's obvious that GLog queries can be translated into the corresponding MapReduce jobs quickly within 1 second, which is negligible for the query time.

TABLE II. Number of Lines of Codes for GLog, Java Programs, Pig and Giraph, and Translation Time for GLog

Query	GLog	Translated Java	Handwritten Java	Giraph	PigLatin	Translation Time
Landmark	11	757	360	233	13	91ms
Reachability	7	591	257	251	10	63ms
Triangle	4	424	375	183	7	12ms
Mutual Neighbors	2	186	150	133	8	15ms
Connected Components	7	657	336	142	13	36ms
PageRank	9	531	250	142	11	56ms
Hub and Authority	13	951	635	208	15	101ms

C. Studies on Optimization Strategies

In this part, we first study the effect of the optimization strategies proposed in Section IV. Then, we compare the query performance of GLog with that of handwritten Java programs.

Figure 8(a) reports the speedup of the fully optimized GLog translated programs over non-optimized ones. The speedup is defined as the ratio of the latter relative to the former with respect to the query evaluation time. X-axis stands for various graph operations in Table II, which are denoted by the first letters for short (the same below). We can observe that the optimized translated programs with fewer MapReduce jobs run up to 2 or 4 times faster than non-optimized ones for operations with iterations. For example, the basic GLog translation method generates 6 MapReduce jobs in one iteration for landmark index construction (denoted by L in X-axis), and it takes 673 minutes to build a landmark index with 15 landmarks on *livejournal*. With the optimizations, we can merge 6 jobs into one job and reduce the indexing time to 165 minutes. As for triangle (denoted by T) and mutual neighbors discovery (denoted by M), the GLog queries are translated in a straightforward way and there is little room for optimizations. These experimental results easily verify our claim before. The reduction of the number of MapReduce jobs is an important optimization strategy in MapReduce framework.

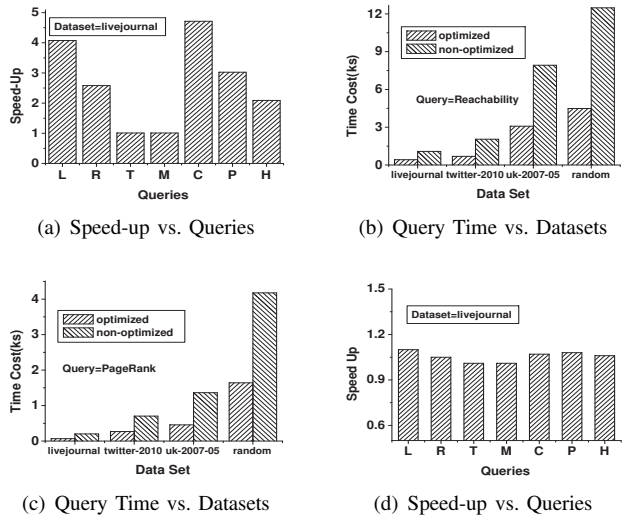
Figure 8(b) and Figure 8(c) study the query evaluation time cost of reachability and PageRank queries respectively over different graphs with optimizations or not. We can obtain the similar results as those in Fig.8(a). For example, we can improve the performance about 2.5 times on *uk-2007-05* for reachability query and achieve a 2-fold speedup per iteration of PageRank.

Figure 8(d) compares the query performance of our optimized translated MapReduce programs and the handwritten programs. We try our best to improve the performance of handwritten ones via strategies like combiner in Hadoop [2] and the reduction of redundant computation cost. In Fig.8(d), we plot the speedup of handwritten programs over optimized GLog ones. We can observe that, with the optimization strategies including rule merging and iteration rewriting, GLog query can achieve nearly the same performance as the well-designed Java programs written by developers. These results indicate that GLog language can not only simplify the query expressions for users, but also yield a quite satisfactory performance.

D. Extensive Studies

In this part, we take the landmark index construction as an example to analyze the scalability of GLog in the Hadoop cluster. Finally, we focus on the performance comparison among GLog, Pig and Giraph.

Figure 9(a) studies the impact of the number of landmark vertices on the indexing performance. We run tests on *livejournal* graph. It is no surprise that the running time goes up with the increase of the number of landmark vertices since we have to locate all shortest paths starting from each landmark vertex. However, the average time cost for a single landmark vertex is going down. For example, the indexing time with 25 landmark vertices is far less than 5 times of that with 5 landmark vertices, which indicates high scalability of GLog queries with the increase of intermediate results over MapReduce.

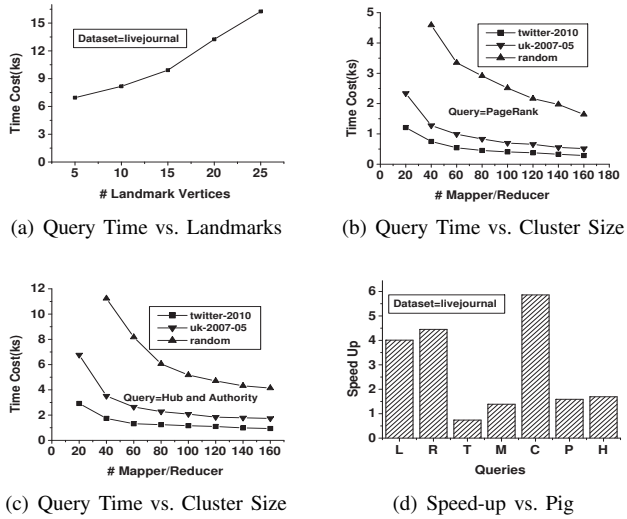


(a) Speed-up vs. Queries (b) Query Time vs. Datasets

(c) Query Time vs. Datasets (d) Speed-up vs. Queries

Fig. 8. Experimental Results on Optimizations

Figure 9(b) and Figure 9(c) show the running time for one iteration in PageRank, hub and authority queries respectively on the Hadoop cluster varying the number of computing nodes over large graphs. We adjust the number of computing nodes by limiting the number of mappers and reducers running concurrently in the cluster. We can see that the evaluation time cost goes down in a nearly linear fashion with the increase of mappers and reducers. It is because each computing node can compute scores for partial vertices independently in PageRank, or hub and authority query. Thus, more computing nodes mean higher parallelism, which yields a better performance. These results also reveal good scalability of GLog queries in terms of computing nodes.



(a) Query Time vs. Landmarks (b) Query Time vs. Cluster Size

(c) Query Time vs. Cluster Size (d) Speed-up vs. Pig

Fig. 9. Experimental Results on Scalability

Next, we compare the performance of GLog query engine with that of Pig. Figure 9(d) plots the speedup of our optimized GLog programs over Pig embedded programs. It is clearly that, for most graph operations, GLog runs faster than Pig and the speedup of GLog over Pig ranges from 1.3 to 5.8. The reasons for the speedup are as follows. GLog is a high-level language tailored to graph operations. It introduces the graph

related features, such as RG table and flexible termination control on iterations. It also attempts to optimize complex MapReduce jobs with iteration aware optimization strategies. In contrast, PigLatin is a general-purpose language. Although it can process the nested data, its support to complex graph operations is weak. For example, we have to embed PigLatin scripts into another programming language, such as Java, to support non-predetermined number of iterations. Consequently, no iteration aware optimization strategies are introduced. Take the connected component discovery as an example. GLog generates only 1 job per iteration, while PigLatin scripts are compiled to 5 jobs.

In addition, we observe that Pig can achieve a similar or even better performance for triangle discovery in Fig.9(d). We collect the translated jobs for Pig and GLog, and discover that these two jobs are similar, both of which contain the equal-join operations for the triangle discovery. We make a further investigation, and find that Pig actually modifies the execution mode of reducer in Hadoop via translated jobs for join operation. We believe that the optimization strategy on underlying Hadoop, rather than the strategy on the query evaluation, contributes to the performance improvement of Pig.

Table III lists comparative results of GLog and Pig on *twitter-2010* and *uk-2007-05*. We first compare the performance between GLog and Pig. We can see the similar results that GLog runs faster on most of operations than Pig, as we have found in Fig.9(d). We also observe an interesting result that the performance comparison is related with the underlying data. For example, Pig runs slower on *twitter-2010* but faster on *uk-2007-05* for hub and authority. It is due to that Pig needs a *FLATTEN* operator to generate the edge table from the nested RG table in order to transfer hub scores along with incoming edges, while the *FLATTEN* is expansive on large bags. In graph *twitter-2010*, there exist vertices with very large degree and then the corresponding edge bags are huge, which degrades the performance of Pig. As for the graph *uk-2007-05*, the maximum degree is less than 16k. Then the *FLATTEN* operator is not the key factor, and the join operation on optimized Hadoop enables Pig to outperform GLog.

We also report the evaluation time of Giraph scripts for the same graph operations in Table III. We find that for graph queries with acceptable intermediate results, Giraph scripts can achieve a better performance than the MapReduce implementations since the former can handle all data in memory. We also notice that Giraph cannot process the triangle discovery or hub and authority queries on given graphs, since Giraph runs out of memory in maintaining the intermediate messages in these two operations. We stress here that GLog is a framework-independent graph query language and has potential to be translated into other distributed frameworks including Giraph.

TABLE III. EVALUATION COST (IN SECONDS)

Query	Graph	Pig	GLog	Giraph
Landmark	*livejournal*	$>$ 6 hours	9915	4021
Reachability	*twitter-2010*	2398	851	173
Reachability	*uk-2007-05*	4083	1380	360
Triangle	*livejournal*	471	641	cannot run
Mutual Neighbors	*twitter-2010*	132	89	52
Mutual Neighbors	*uk-2007-05*	155	101	71
Mutual Neighbors	*random*	633	513	398
PageRank	*twitter-2010*	422	286	33
PageRank	*uk-2007-05*	550	516	81
PageRank	*random*	1982	1642	726
Hub and Authority	*twitter-2010*	1070	908	54
Hub and Authority	*uk-2007-05*	1480	1719	168
Hub and Authority	*random*	4028	4136	cannot run

E. Summary

We can draw the following conclusions from the experimental results: i) GLog is a succinct high level language and can be used in various graph analysis tasks; ii) The optimization strategies to reduce the number of MapReduce jobs can improve the performance of GLog queries significantly; iii) GLog query engine on Hadoop can achieve a better performance than Pig for most of graph operations, and deliver good scalability in terms of computing nodes in Hadoop cluster. iv) Giraph can achieve a better performance for graph queries with acceptable intermediate results, but cannot handle queries with massive intermediate results. As a complementarity to the MapReduce framework, Giraph will serve as an interesting underlying framework for GLog in the future.

VI. RELATED WORK

In the following, we review the state-of-the-art techniques and point out why they cannot fully support the high level graph analysis using distributed frameworks.

Graph Query Processing. Graph pattern query finds a given target sub-graph in a data graph [13]. It can be expressed in a relatively simple way, as developers only need describe the structure of target sub-graphs. However, the functionality of graph pattern query is limited, as it cannot express other operations like the minimal spanning tree construction, PageRank [6], connected components discovery, etc.

There are also studies on general-purpose graph operations, such as graph algebra [11]. Similar to relational algebra, graph algebra consists of a collection of operations, including selection, projection, concatenation, etc. We can locate a specific sub-graph in a data graph or even transform a graph using graph algebra. However, the expression of graph algebra is complex. It takes a nested form, as the structure of a transformed graph is complex and flexible. In addition, the current graph algebra operations do not support recursion or iteration, which is needed by many graph operations.

High Level Graph Query Language. As a graph can be represented by an edge table, we can use Datalog to express graph operations, like the reachability query, or even various network routing protocols [7] on the edge table. Datalog is a high level query language which runs on the relational data originally [24], and it can support recursion naturally compared with relational algebra.

The evaluation of Datalog queries in distributed platforms is studied [9]. Due to the high cost incurred by a large number of iterations in distributed platforms, they propose an approach to rewrite the query into a non-linear Datalog query, which needs much fewer iterations. However, the queries which can be optimized in such a way are limited, and there are many other cases which cannot be rewritten (*e.g.,* the query which finds whether a vertex u can reach another vertex v).

Socialite [16] is closely related to our work. It uses Datalog in social network analysis, takes semi-native evaluation and

chooses an evaluation order to optimize the query. However, it is hard for developers to incorporate optimization strategies on graph operations using Socialite, such as the selection of frontier nodes in the landmark index construction. More importantly, Socialite does not discuss the implementation and optimization in distributed frameworks.

High Level Distributed Data Processing System. The high level data processing language is not new in distributed platforms. As most of developers are familiar with SQL language, many systems provide a SQL-style query language over the structured data in Hadoop platform. Hive [5] is one representative system, which can translate a SQL like language into a sequence of MapReduce jobs. YSmart system [21] is developed to handle the input correlation and transit correlation. Fewer MapReduce jobs are generated to lower the redundant cost for complex queries, which makes YSmart faster. Pig [8] is another high level data flow system, which also generates MapReduce jobs from user-defined scripts.

However, these systems are not suitable to general graph queries. Hive [5] and YSmart [21] mainly handle structured data. Although Pig can manipulate nested data, Piglatin [8] provides a weak support to iterations. It has "foreach" control statement, but cannot enable developers to express dynamic termination conditions explicitly. Then PigLatin cannot offer optimization strategies over iterative MapReduce jobs.

In summary, none of the existing systems can address the high level graph analysis ideally. First, there does not exist a widely-accepted, high level graph analysis language which can balance the simplicity and expressiveness. Second, the translation and optimization of such a language in distributed frameworks will face more challenges than other existing high level query systems [5], [21], [8], as most of graph operations incur costly iterative jobs.

VII. Conclusions and Future Work

In this paper, we introduce RG tables specific to graph data, and design its high level graph analysis language GLog. GLog shares the similarity to Datalog for simplicity, and has its explicit iteration control statement for expressiveness. We also develop translation templates for GLog rules into MapReduce jobs, and devise iteration aware optimization strategies on the translated jobs. The experimental evaluation shows that GLog can not only express various graph analysis tasks in a succinct way, but also achieve a quite satisfactory performance for various graph analysis tasks.

The work can be extended in the following interesting ways. First, we can enhance the expressive power of GLog and its underlying RG tables. For example, dynamic number of vertices will be allowed in one RG table to support more graph queries. At the same time, we require that GLog query remains simple. Second, we attempt to migrate the work from the MapReduce framework to other frameworks, such as Pregel-like systems. In a long run, we expect GLog queries can be evaluated on different frameworks, and the optimizer in the query engine will choose a proper framework for queries with different requirements transparently.

Acknowledgment

This work was supported by NSFC under Grant No. 61073018 and 61272156, Research Grants Council of the Hong Kong SAR, China No. 418512, National High Technology Research and Development Program of China under No. 2012AA011002.

References

[1] *Apache Giraph*. http://incubator.apache.org/giraph/.

[2] *Apache Hadoop*. http://hadoop.apache.org.

[3] *Apache Pig*. http://pig.apache.org/.

[4] A.Gates, O.Natkovich, S.Chopra, and et.al. Building a high level dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[5] A.Thusoo, J.Sarma, N.Jain, and et al. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[6] B.Bahmani, K.Chakrabarti, and D.Xin. Fast personalized pagerank on mapreduce. In *SIGMOD*, pages 973–984, 2011.

[7] B.Loo, T.Condie, M.Garofalakis, and et al. Declarative networking: language, execution and optimization. In *SIGMOD*, pages 97–108, 2006.

[8] C.Olston, B.Reed, U.Srivastava, R.Kumar, and A.Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[9] F.Afrati and J.Ullman. Transitive closure and recursive datalog implemented on clusters. In *EDBT*, pages 132–143, 2012.

[10] G.Malewicz, M.H.Austern, A.J.C.Bik, and et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[11] H.He and A.K.Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418, 2008.

[12] H.Yang, A.Dasdan, R.Hsiao, and D.Jr. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.

[13] J.Cheng, J.Yu, B.Ding, P.Yu, and H.Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.

[14] J.Dean and S.Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[15] J.Huang, D.Abadi, and K.Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

[16] J.Seo, S.Guo, and M.Lam. Socialite: Datalog extensions for effcient social network analysis. In *ICDE*, 2013.

[17] K.Beyer, R.Cochrane, V.Josifovski, and et.al. System rx: One part relational, one part xml. In *SIGMOD*, pages 347–358, 2005.

[18] K.Haewoon, L.Changhyun, P.Hosung, and M.Sue. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[19] M.Potamias, F.Bonchi, C.Castillo, and A.Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 453–470, 2009.

[20] P.Boldi, M.Santini, and S.Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[21] R.Lee, T.Luo, Y.Huai, F.Wang, Y.He, and X.Zhang:. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, pages 25–36, 2011.

[22] S.Blanas, J.Patel, V.Ercegovac, and et al. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986, 2010.

[23] U.Kang, C.E.Tsourakakis, and C.Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, pages 229–238, 2009.

[24] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*. 1988.

[25] Y.Low, J.Gonzalez, A.Kyrola, and et al. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[26] Y.Tian, S.Tatikonda, and B.Reinwald. Scalable and numerically stable descriptive statistics in systemml. In *ICDE*, pages 1351–1359, 2012.