Pregel meets UnCAL: a Systematic Framework for Transforming Big Graphs

Le-Duc Tung
The Graduate University for Advanced Studies
Shonan Village, Hayama, Kanagawa 240-0193, Japan
Email: tung@nii.ac.jp
Expected graduation date: March 2016
Supervised by Zhenjiang Hu

***Abstract*—Graph is a multi-purpose tool to represent many different kinds of data from tranditional datasets to social networks. At present, *Pregel* is a popular graph computation model to deal with big graphs up to billion vertices and trillion edges. However, Pregel programming model is very low-level and requires developers to write programs that are hard to maintain and need careful optimizations. In this thesis we are developing *Gito*, a systematic framework on top of Pregel to do transformations over big graphs. Transformations in Gito are expressed in a SQL-like language - UnQL - whose internal algebra is UnCAL, and then are compiled into Pregel code. In particular, in this paper, we show the feasibility of integrating UnCAL and Pregel, and propose a scalable Pregel-based computation for a subclass of UnCAL. Our preliminary results are encouraging and allow us to go further for a complete framework.**

I. Introduction

Pregel [1] is a new computation model proposed by Google to process big graphs. It was inspired by the *Bulk Synchronous Parallel* computation model in which the whole computation consists of a sequence of *supersteps*. During a superstep, every vertex executes exactly once a common function defined by *Vertex*.**compute**(). Inside *Vertex*.**compute**(), a vertex can receive messages from the previous superstep, do its computation (i.e. updating their values, adding/removing edges and vertices), and send messages to other vertices in the next superstep. Changes in vertices and edges will be available in the next superstep. Interleaving with supersteps are barriers in order to synchronize communication data.

Although Pregel is scalable to very large graphs up to billion vertices and trillion edges, its programming model is very low-level in the sense that the whole computation is condensed in a single function *Vertex*.**compute**(), resulting in very long and complex programs. Consequently, there are many optimizations for specific Pregel-based algorithms. Similar issues have been raised for map-reduce model, leading to high-level query languages such as *HiveQL* or *PigLatin* on top of the map-reduce model. Inspired by *Hive/Pig* systems, we aim to build a high-level framework on top of Pregel model.

We choose UnQL [2] as a front-end language for our framework due to two important reasons. First, since UnQL is a SQL-like query language designed for querying semistructure data, users in database community do not need to learn a completely new language. Second, UnQL consists of an optimizable algebra - UnCAL, in which UnQL queries are translated to structural recursions to which many rewriting rules can be applied systematically. However, despite of being powerful, UnCAL has problems of scalability. Experiments in [2] on a sequential version of UnCAL only dealt with graphs up to 10 thousand nodes. Suciu [3] proposed a distributed evaluation with a setting of graph partitions, but potentially, for big graphs, there existed a bottleneck because its reachability computation was done at a single site [4].

Motivated by the advantages of both Pregel and UnCAL (the core of UnQL), we propose a systematic framework on top of Pregel model to do graph transformations, which allows users to express transformations in a SQL-like query language - UnQL. UnQL queries are then compiled into a scalable Pregel program. Such a framework has some advantages. On the user side, they do not need to have knowledge of how Pregel works, but still obtain an efficient and scalable implementation automatically. On the system side, we can systematically apply many optimization rules to rewrite queries.

Our approach is to explore the potential of parallelism in UnCAL and to point out bottlenecks that limit its scalability. We show a subclass of UnCAL that can be potentially paralellized and then propose a scalable computation on top of Pregel. Our contribution so far is a scalable framework that can handle a subclass of UnQL. This contribution is the first important step towards a complete framework that deals with arbitrary UnQL queries.

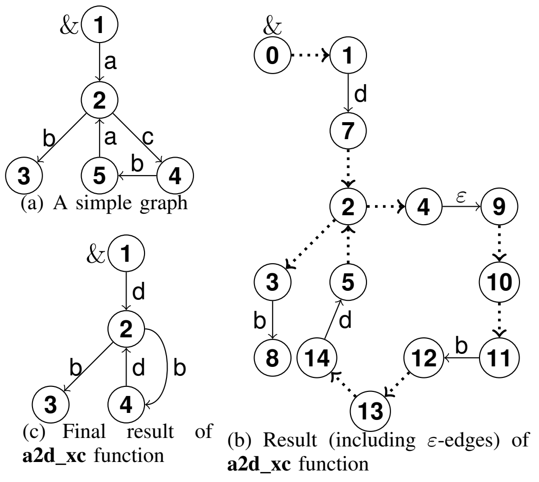
II. UnCAL : a “λ-calculus” for Graphs

In this section, we briefly review the UnCAL (Unstructured CALculus), a powerful graph algebra [2] that functions as a core component in our framework.

A. Graph Data Model

UnCAL's data model is a directed edge-labeled graph extended by *markers* and *ε-edges* [2]. Edge-labeled graphs are in the sense that data are stored on edges, while vertices are unique identity objects without labels. Markers are symbols to designate certain vertices as *input vertices* or *output vertices*. ε-edges are edges labeled with a special symbol ε. Such ε-edges could be thought as an “empty” transition in automata.

Let $Label$ be a set of labels, $\mathcal{M}$ be an infinite set of markers denoted by $\&x$, $\&y$, $\&z$, ... There is a distinguished marker $\& \in \mathcal{M}$ called a *default marker*.


(a) A simple graph

(c) Final result of **a2d_xc** function

(b) Result (including ε-edges) of **a2d_xc** function

Fig. 1: Examples of Rooted Edge-labeled Graphs

Definition 1 (Graph with Markers [2]). A graph G is a quadruple (V, E, I, O), where V is a set of vertices, $E \subseteq V \times \{\mathit{Label} \cup \varepsilon\} \times V$ is a set of edges, $I \subseteq \mathcal{M} \times V$ is an one-to-one mapping from a set of input markers to V, and $O \subseteq V \times \mathcal{M}$ is a many-to-many mapping from V to a set of output markers.

For $\&x, \&y \in \mathcal{M}$, let $v = I(\&x)$ be the unique vertex such that $(\&x, v) \in I$, we call v an *input vertex*. If there exists a $(v, \&y) \in O$, we call v an *output vertex*. Note that there are no edges coming to input vertices or leaving from output vertices. Let $DB^{\mathcal{X}}_{\mathcal{Y}}$ denote data graphs with sets of input markers $\mathcal{X}$ and output markers $\mathcal{Y}$. When $\mathcal{X} = \{\&\}$, $DB^{\mathcal{X}}_{\mathcal{Y}}$ is abbreviated to $DB_{\mathcal{Y}}$, and $DB_{\emptyset}$ is abbreviated to DB. *A rooted graph* is the one that has only one input marker $\mathcal{X} = \{\&\}$ and no output markers $\mathcal{Y} = \emptyset$, in which the vertex $v = I(\&)$ is called the root vertex of the graph. Graphs with multiple markers are internal data structures for graph constructors.

Figure 1(a) shows an example of a rooted directed edge-labeled graph in which $V = \{1, 2, 3, 4, 5\}$, $E = \{(1, a, 2), (2, b, 3), (2, c, 4), (4, b, 5), (5, a, 2)\}$, $I = \{(\&, 1)\}$, and $O = \{\}$. The vertex with identity 1 is the root of the graph, and is marked with &.

B. Graph Constructors

Before looking at graph constructors in details, we need to define an additional operation $\cdot$ to generate new markers. The operation $\cdot$ returns a different marker for every pair of $\&x$ and $\&y$. We assume $\cdot$ to be associative, $(\&x \cdot \&y) \cdot \&z = \&x \cdot (\&y \cdot \&z)$, and $\&$ to be its identity, $\& \cdot \&z = \&z \cdot \& = \&z$.

There are nine graph constructors in UnCAL. From these constructors, we can systematically build arbitrary edge-labeled graphs. Definitions of the constructors are given in Fig. 2 (Dotted arrows denote ε-edges). Informally, $\{\}$ constructs a graph of only one vertex labeled with default input marker $\&$, $\{l : G\}$ contructs a new graph G' from the graph G by adding the edge l pointing to the root of G. The source vertex of l becomes the root of G'. The operator $\cup$ unions two graphs of the same input markers with the aid of ε-edges. The next two constructors allow us to add input and output markers: $\&z := G$ takes a graph $G \in DB^{\mathcal{X}}_{\mathcal{Y}}$ and relabels input vertices with the input marker $\&z$, thus the result is in $DB^{\mathcal{Z}\cdot\mathcal{X}}_{\mathcal{Y}}$; $\&y$ returns a graph of a single vertex labeled with the default input marker $\&$ and the output marker $\&y$. $()$ contructs an empty graph without any markers and vertices. The disjoint union $G_1 \oplus G_2$ requires two graphs G_1 and G_2 have disjoint sets of input markers. The operator $G_1 \mathbin{@} G_2$ vertically constructs a graph by plugging output markers of G_1 to input markers of G_2. It requires $G_1 \in DB^{\mathcal{X}}_{\mathcal{Y}}$ and $G_2 \in DB^{\mathcal{Y}}_{\mathcal{Z}}$, thus $G_1 \mathbin{@} G_2 \in DB^{\mathcal{X}}_{\mathcal{Z}}$. Finally, the last operator allows us to introduce cycles by adding ε-edges from an output marker to the input marker named after it.

$$
\begin{array}{rll}
e ::= & \{\} \mid \{l : e\} \mid e \cup e \mid \&x := e \mid \&y \mid () & \\
\mid & e \oplus e \mid e \mathbin{@} e \mid \mathbf{cycle}(e) & \{\text{constructors}\} \\
\mid & \$g & \{\text{graph variable}\} \\
\mid & \mathbf{if}\ l = l\ \mathbf{then}\ e\ \mathbf{else}\ e & \{\text{conditional}\} \\
\mid & \mathbf{rec}(\lambda(\$l, \$g).e)(e) & \{\text{structural recursion application}\} \\
l ::= & a \mid \$l & \{\text{label } (a \in \mathit{Label}) \text{ and label variables}\}
\end{array}
$$

Fig. 3: Core UnCAL Language

Example 1. The graph in Fig. 1(a) can be constructed as follows. (but not uniquely)

$$
\begin{aligned}
\&z \mathbin{@} \mathbf{cycle}((&\&z := \{\mathrm{a} : \&z_1\}) \\
&\oplus (\&z_1 := \{\mathrm{b} : \{\}\} \cup \{\mathrm{c} : \{\mathrm{b} : \&z_2\}\}) \\
&\oplus (\&z_2 := \{\mathrm{a} : \&z_1\})) \qquad \Box
\end{aligned}
$$

For brevity, we write $\{l_1 : G_1, \ldots, l_n : G_n\}$ to denote $\{l_1 : G_1\} \cup \ldots \cup \{l_n : G_n\}$, and $(G_1, \ldots, G_n)$ to $G_1 \oplus \ldots \oplus G_n$.

C. UnCAL Syntax

Figure 3 depicts UnCAL's core syntax that consists of nine graph constructors, variables, conditionals, and structural recursion. The graph constructors have already been mentioned before, while variables and conditionals are self explanatory. Hence, we focus on *structural recursion*, a powerful mechanism to express transformations on graphs.

A function f on graphs is called a structural recursion if it is defined by the following equations

$$
\begin{array}{lcl}
f(\{\}) & = & \{\} \\
f(\{\$l : \$g\}) & = & e(\$l, \$g) \mathbin{@} f(\$g) \\
f(\$g_1 \cup \$g_2) & = & f(\$g_1) \cup f(\$g_2)
\end{array}
$$

The first and the third equation are always in that form, so we encode f as $\mathbf{rec}_{\mathcal{S}}(\lambda(\$l, \$g).e)$ where e is of type $DB^{\mathcal{S}}_{\mathcal{S}}$, sometimes for brevity, we just write $\mathbf{rec}(e)$. It is interesting that though the structural recursion form is quite simple, it is very powerful to describe many graph transformations including all graph queries (in UnQL) [2], and non-trivial model transformations.

Example 2. The following structural recursion **a2d_xc** relabels edges a to d and contracts edges c. Applying this function to the graph in Fig. 1(a) results the graph in Fig. 1(c).

$$
\begin{aligned}
\mathbf{a2d\_xc}(\$db) = \mathbf{rec}(\lambda(\$l, \$g).\ & \\
&\mathbf{if}\ \$l = \mathrm{a}\ \mathbf{then}\ \{\mathrm{d} : \&\} \\
&\ \mathbf{else\ if}\ \$l = \mathrm{c}\ \mathbf{then}\ \{\varepsilon : \&\} \\
&\quad \mathbf{else}\ \{\$l : \&\})(\$db) \qquad \Box
\end{aligned}
$$

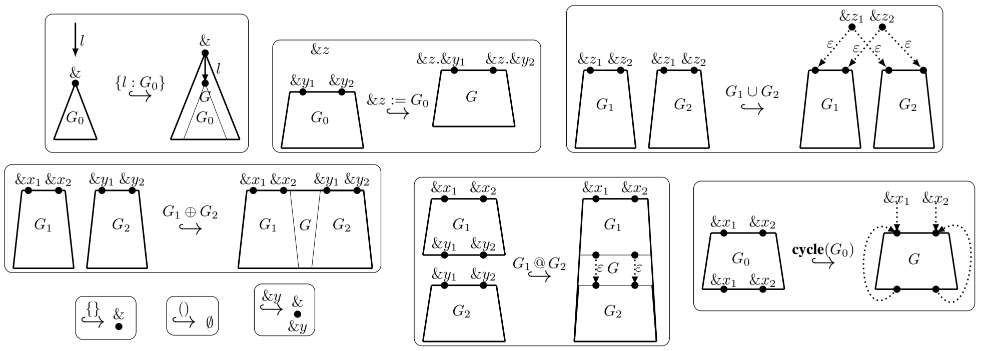


Fig. 2: Graph Constructors

III. Parallel Evaluation of Structural Recursion

One of the advantages of structural recursion is the ability of composing multiple structural recursions to describe many complex transformations, i.e. $\textbf{rec}(e_2) \circ \textbf{rec}(e_1)$. However, compositions usually lead to large intermediate graphs or multiple graph traversals. Thanks to tupling and fusion rules, we can solve those problems systematically by rewriting multiple structural recursions into one structural recursion [2], [5], [6]. Therefore, if we can parallelize a structural recursion and make it scalable on Pregel, then we can achieve the scalability of many graph transformations.

A. Parallelizable Structural Recursion

In this section, we show how systematically evaluate a class of structural recursion that can be efficiently computed in parallel/distributed way. It produces the same result as the bulk semantics of structural recursion does in [2].

Proposition 1 ([2])**.** *If the expression $e(\$l, \$g)$ in $\textbf{rec}_{\mathcal{Z}}(e)$ does not depend on the graph variable $\$g$, then the following data value equalities hold*

$$
\begin{aligned}
\textbf{rec}(e)(\{\}) &\equiv (\&z_1 := \{\}, \ldots, \&z_n := \{\}) \\
\textbf{rec}(e)(\{\$l : \$g\}) &\equiv e(\$l) \mathbin{@} \textbf{rec}(e)(\$g) \\
\textbf{rec}(e)(\$g_1 \cup \$g_2) &\equiv \textbf{rec}(e)(\$g_1) \cup \textbf{rec}(e)(\$g_2) \\
\textbf{rec}(e)(\&x := \$g) &\equiv \&x \cdot \textbf{rec}(e)(\$g) && (1) \\
\textbf{rec}(e)(\&y) &\equiv (\&z_1 := \&y \cdot \&z_1, \ldots, \&z_n := \&y \cdot \&z_n) && (2) \\
\textbf{rec}(e)() &\equiv () \\
\textbf{rec}(e)(\$g_1 \oplus \$g_2) &\equiv \textbf{rec}(e)(\$g_1) \oplus \textbf{rec}(e)(\$g_2) \\
\textbf{rec}(e)(\$g_1 \mathbin{@} \$g_2) &\equiv \textbf{rec}(e)(\$g_1) \mathbin{@} \textbf{rec}(e)(\$g_2) && (3) \\
\textbf{rec}(e)(\textbf{cycle}(\$g)) &\equiv \textbf{cycle}(\textbf{rec}(e)(\$g)) && (4)
\end{aligned}
$$

In Eq.(1), $\&x \cdot (\&z_1 := \$g_1, \ldots, \&z_n := \$g_n)$ denotes $(\&x \cdot \&z_1 := \$g_1, \ldots, \&x \cdot \&z_n := \$g_n)$. We call the function $\textbf{rec}(e)$ a parallelizable structural recursion.

This proposition suggests a divide-and-conquer computation in which we apply the function e on every edge of the input graph, then constructively build a result graph. This graph, however, contains a lot of ε-edges due to graph constructors, and we basically eliminate these ε-edges by computing their transitive closure, which is a time-comsuming task. Also note that this divide-and-conquer computation is natural in dealing with cycles as well.

Now, let us see how to use the above equatilies to evaluate the structural recursion **a2d_xc**. Note that we have already rewritten the input graph by constructing it from each individual edge.

$$
\begin{aligned}
&\textbf{rec}_{\{\&\}}(e)(\&z \mathbin{@} \textbf{cycle}((\&z := \{\mathrm{a} : \&z_1\}, \\
&\qquad \&z_1 := \{\mathrm{b} : \{\}\} \cup \{\mathrm{c} : \&z_2\}, \\
&\qquad \&z_2 := \{\mathrm{b} : \&z_3\}, \\
&\qquad \&z_3 := \{\mathrm{a} : \&z_1\}))) \\
&\equiv \{\text{ Eq. (3), } @ \text{operator; Eq. (4), promoting } \textbf{rec}\} \\
&(\& := \& \cdot \&z) \mathbin{@} \textbf{cycle}((\textbf{rec}_{\{\&\}}(e)(\&z := \{\mathrm{a} : \&z_1\}), \\
&\qquad \textbf{rec}_{\{\&\}}(e)(\&z_1 := \{\mathrm{b} : \{\}\} \cup \{\mathrm{c} : \&z_2\}), \\
&\qquad \textbf{rec}_{\{\&\}}(e)(\&z_2 := \{\mathrm{b} : \&z_3\}), \\
&\qquad \textbf{rec}_{\{\&\}}(e)(\&z_3 := \{\mathrm{a} : \&z_1\}))) \\
&\equiv \{\text{ Eq. (1); Eq. (2); } \textbf{rec}(\mathrm{e}) = \textbf{a2d\_xc} \text{ on edges}\} \\
&(\& := \& \cdot \&z) \mathbin{@} \textbf{cycle}((\& \cdot \&z := \{\mathrm{d} : \&\} \mathbin{@} (\& := \& \cdot \&z_1), \\
&\qquad \& \cdot \&z_1 := \{\mathrm{b} : \{\}\} \cup (\{\varepsilon : \&\} \mathbin{@} (\& := \& \cdot \&z_2)), \\
&\qquad \& \cdot \&z_2 := \{\mathrm{b} : \&\} \mathbin{@} (\& := \& \cdot \&z_3), \\
&\qquad \& \cdot \&z_3 := \{\mathrm{d} : \&\} \mathbin{@} (\& := \& \cdot \&z_1)))
\end{aligned}
$$

Intuitively, Fig. 1(b) shows the graph produced by the above procedure. The Fig. 1(c) is the final result after eliminating ε-edges from the graph in Fig. 1(b).

B. Pregel Implementation of Parallelizable Structural Recursions

As shown above, evaluating a structural recursion $\textbf{rec}(e)$ basically includes two phases: the first to apply the function e on each edge and the second to eliminate ε-edges.

The first phase seems to be scalable in Pregel, but it actually causes a serious bottleneck at the root vertex. Because our data model is a rooted graph, every data object will be plugged to the root (i.e. a rooted graph for storing 1 million papers would have 1 million outgoing edges labeled Paper from the root). During the first phase, a huge number of new edges emanating from the root are created and the root must take care of those edges (processing and storing them), leading to poor scalability. Our idea is avoiding a centralized computation at the root by creating virtual roots. We create new ε-edges (r, v) from the root r, and one data object will emanate from one vertex v instead of r. Let us consider v as virtual roots, evaluation thus starts from vertices v instead of r, and properties of r are reserved to v. Technically, one can consider this procedure as a duplication of the root vertex. But, in theory, we need ε-edges to make graphs bisimilar. After finishing the whole evaluation, we will merge these virtual roots into a single root of a result graph. Here, we need just one superstep for the first phase.

The second phase is divided into three sub-phases. Each sub-phase requires multiple supersteps. First, we compute reachability from the root, then remove all vertices and edges that are unreachable from the root, and finally contract ε-edges. Theoretically, an ε-edge from vertex v to v' means that all edges going out from v' should be going out from v. Therefore, contracting an ε-edge (v, v') means removing this ε-edge and for each edge (v', a, w), a new (v, a, w) is added.

Again, we detected another bottleneck during this ε-edge contraction. We observed that although virtual roots initially have just one edge, after a number of supersteps of contraction, these virtual roots have to create (and store) a huge number of edges so that the contraction cannot finish. The reason is as follows. After applying the function e (the first phase), virtual roots point to ε-edges, these ε-edges also go to another ε-edges, resulting in a tree-like graph in which internal edges are labeled by ε and its "leaf edges" are non-ε edges. These leaf edges finally are added to virtual roots due to the contraction, causing a bottleneck there. This problem is typical in the case of queries having regular expressions starting with $\_*$ (i.e. $\{\_*.\mathrm{Paper.Title}\}$). Our solution is, instead of contracting these ε-edges, we propagate virtual nodes along ε-edges and finally promote source vertices of leaf edges to be virtual roots. This solution has two advantages. First, we do not need to create new edges during the contraction. Second, this solution can be integrated to the reachability phase easily.

We summarize our Pregel implementation by partially showing a source code of the function *Vertex*.**compute**() which applies to each vertex (Listing 1).

```
public void compute(
    Vertex<LongWritable,BoolIntWritable,T> vertex,
    Iterable<LongWritable> messages)
    throws IOException {
  VertexWorkerContext<T> workerc = getWorkerContext();
  switch (workerc.getPhase()) {
  case SR_APPLY_FUNCTION: /* rec(e) on {l:g} */
     applySRFunction(vertex, workerc);
     break;
  case SR_COMP_REACHABILITY_ROOT_PROPAGATION:
     computeReachAndRootProp(vertex,messages);
     break;
  case SR_REMOVE_REDUNDANT_PARTS:
     if (isSingleVtx(vertex) || isUnreach(vertex)){
       removeVertexRequest(vertex.getId());
     }
     break;
  case SR_CONTRACT_EPS:
     contractEpsEdges(vertex, messages);
     break;
  case SR_FINISH:
     break;
  default:
     break;
  }
}
```

Listing 1: The Function **compute**() of a Vertex in Gito

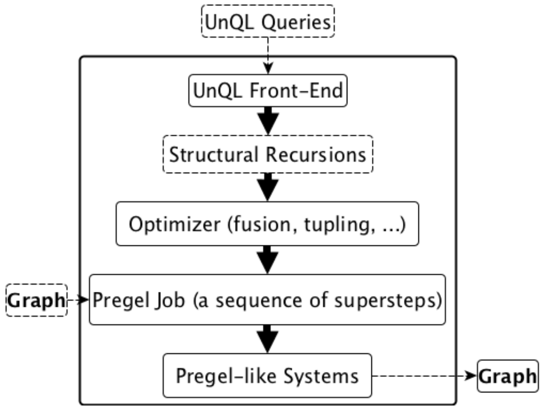


Fig. 4: Gito Architecture

IV. GITO: A SYSTEMATIC FRAMEWORK OVER PREGEL-LIKE SYSTEMS

A. *Gito Architecture*

We propose Gito[1] (Graph in Tokyo) - a systematic framework over Pregel model. The architecture of Gito is shown in Fig. 4. Gito accepts a rooted edge-labeled graph as its input and produces a new rooted edge-labeled graph. Users do transformations by specifying UnQL queries or structural recursions (in UnCAL). Components in Gito are as follows.

- **UnQL Front-End**: parsers input UnQL queries and transforms them to structural recursions.
- **Optimizer**: rewrites structural recursions to produce a single structural recursion.
- **Pregel Job**: implements a structural recursion **rec**(e).
- **Pregel-like Systems**: open-source implementations of Google Pregel computation model.

B. *Preliminary Results*

We used Apache Giraph[2] as an open-source implementation of Google Pregel computation model for our framework. We borrowed citation network datasets of DBLP and ACM papers[3]. Each paper is associated with abstract, authors, year, venue, and title. To make the balance between edge labels, we temporarily removed abstracts from papers. In our experiments, we used the dataset Citation-network-V1 which consists of $629,814$ papers from year 1941 to 2010 and more than $632,752$ citations. A rooted edge-labeled graph of that dataset includes $8,252,753$ vertices and $11,427,500$ edges. We did experiments on a small cluster of four machines: each one has 16 cores and 32GB RAM, network speed 56Gb/s. However, we just used 60 cores in total, and reserved four cores (one in each machine) to do administrator tasks.

$$\begin{aligned}Q_1 = \textbf{select}\ & \\ & (\textbf{select}\ \{\mathrm{Scientist} : \$s\} \\ & \quad \textbf{where}\ \{\_*.\mathrm{Name} : \$s\}\ \textbf{in}\ \$t) \\ & \textbf{where}\ \{\_*.\mathrm{Author} : \$t\}\ \textbf{in}\ \$db\end{aligned}$$

[1]Source code: http://www.prg.nii.ac.jp/members/tungld/gito-Dec2014.tar.gz.
[2]Giraph:http://giraph.apache.org.
[3]Citation network datasets: http://arnetminer.org/citation.

TABLE I: The query Q_1 using 60 workers (60 cores)

# of edges (in million)	2	4	6	8	11 (the whole graph)
# of supersteps	18	18	18	18	18
Running time[4] (in second)	27	30	34	38	44

TABLE II: The query Q_1 on the whole graph

# of workers	10	20	30	40	50	60
Running time[4] (in second)	103	59	50	42	42	44

The above UnQL query Q_1 was used in our experiments. It finds all authors in the input graph and returns author's names binding with new edges labeled by $\mathrm{Scientist}$.

Table I shows the scalibility of our framework. We set the number of workers to 60 and change the size of graph from 2 million edges to 11 million edges (the whole graph). Here, our framework computes Q_1 in 18 supersteps. The result shows that our framework is very scalable. The running time for the whole graph is just 1.6 times slower than the one for the graph of 2 million edges. We also see that when we increase the size of graphs, the running time seems to be increased a bit faster. This is because there is a dramatic increase of citations between papers. Larger graphs consist of newer papers that have many references to older ones.

Now, we consider the performance of our framework. We process the whole graph, and change the number of workers. Table II shows the result. The speedup nearly doubles when we double the number of workers from 10 to 20. However, after that, the speedup is not so high due to an increase in communication cost. The function e of a structural recursion **rec**(e) which applies on each edge will produce new vertices and edges. Therefore, Giraph requires a lot of communication to reorganize new vertices and edges. In other words, there is a tradeoff between the number of workers and communication cost in our framework.

V. Remaining Works

Currently, our framework works well with restricted *select-where* queries in UnQL [3] that can be translated to mutually parallelizable structural recursions. In the future, we want to deal with more complex queries such as queries with existential conditions, queries with joins. Basically, those complex queries lead to structural recursions **rec**(e) whose function e depends on the graph variable $\$g$. It means that, to evaluate the function e on one edge, we have to explore the other parts of the input graph, in the worst case, the whole graph. Some theoretical works give solutions for some particular cases. Hidaka et. al. [6] shows that there is a class of UnCAL in which the dependence of the function e on the graph varialbe $\$g$ can be eliminated by a static analysis. Suciu [3] shows queries with existential conditions can be evaluated by a restricted *select-where* query plus an additional query. However, we need further analyses to see whether they are suitable for Pregel model or not.

Efficient computation of transitive closure (TC) in Pregel environment is also an important improvement to our framework. Afrati et. al. [7] proposes an efficient distributed algorithm to compute TC on clusters. That would be interesting to integrate that algorithm into our framework to eliminate ε-edges.

[4]in which Giraph took about 15 seconds for input, setup and shutdown

VI. Related Work

UnQL together with its internal algebra UnCAL has been proposed for a long time. Unfortunately, there are not many practical works on these, especially for big graphs. Experiments in [2] only dealt with graphs up to 10 thousand nodes. Suciu [3] proposed a distributed evaluation with the setting of graph partitions, but potentially there is a bottleneck when dealing with big graphs [4]. GraphQL [8] is also a graph query language whose core is a graph algebra. It considers graphs as a basic unit in the query, which is not familiar as a SQL-like query. Moreover, scalability to very large graph databases is an open problem to GraphQL [8]. One of the few works on processing queries using Pregel is proposed by Nole et. al. [9], in which Brzozowski's derivation of regular expressions are exploited. In consequence, queries are limited to regular path queries.

VII. Conclusion

Graph database is very powerful to express many kinds of datasets and to expose important relationships amongs them. We have proposed a systematic framework for transformations over big graphs. Our approach is to combine the advantages from a solid foundation of graph algebra and a practical scalable graph processing model. Preliminary results showed that this combination is very promising when we achieve both good scalability and speedup. In the future we will strengthen the framework to support more complex queries, bridging the gap between UnCAL and big graphs.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, 2010.

[2] P. Buneman, M. Fernandez, and D. Suciu, "Unql: A query language and algebra for semistructured data based on structural recursion," *The VLDB Journal*, vol. 9, no. 1, Mar. 2000.

[3] D. Suciu, "Distributed query evaluation on semistructured data," *ACM Trans. Database Syst.*, vol. 27, no. 1, Mar. 2002.

[4] L.-D. Tung, Q. Nguyen-Van, and Z. Hu, "Efficient query evaluation on distributed graphs with hadoop environment," in *Proceedings of the Fourth Symposium on Information and Communication Technology*, ser. SoICT '13. New York, NY, USA: ACM, 2013.

[5] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu, "A query language and optimization techniques for unstructured data," *SIGMOD Rec.*, vol. 25, no. 2, Jun. 1996.

[6] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, K. Nakano, and I. Sasano, "Marker-directed optimization of uncal graph transformations," in *Proceedings of the 21st International Conference on Logic-Based Program Synthesis and Transformation*, ser. LOPSTR'11, 2012.

[7] F. N. Afrati and J. D. Ullman, "Transitive closure and recursive datalog implemented on clusters," in *Proceedings of the 15th International Conference on Extending Database Technology*, ser. EDBT '12, 2012.

[8] H. He and A. K. Singh, "Graphs-at-a-time: Query language and access methods for graph databases," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, 2008.

[9] M. Nolé and C. Sartiani, "Processing regular path queries on giraph," in *EDBT/ICDT Workshops*, 2014.