Processing UnQL Graph Queries with Pregel

Le-Duc Tung, Chong Li, Xiaodong Meng, Zhenjiang Hu

Pregel is a programming model proposed by Google to process large graphs. It is inspired by the Bulk Synchronous Parallel model by which the same computation instructions are applied to every vertex in each iteration. However, Pregel is a so-called low-level model for end users, and it requires much effort on writing efficient programs, even to well-known graph algorithms. Similar problems with the MapReduce model have resulted in high-level query frameworks such as Hive or Pig on top of MapReduce. Taking the same philosophy, in this paper, we propose a high-level framework on top of Pregel to allow executing queries and transformations over large graphs. We borrow UnQL, an SQL-like language over graphs, as the interface language for our framework. UnQL queries are then automatically compiled into efficient Pregel programs that can deal with large graphs. Experimental results with real-life graphs such as citation networks, Amazon products and Youtube, show that our framework is efficient and scalable for large graphs.

1 Introduction

Data become more and more complex today. Social networks such as Facebook, Twitter, and LinkedIn now have billions of active users [19], and new connections among users are increasing day by day. A world-wide-web network might contain billions of websites and trillions of links among them, where each of those websites does not conform to any standard structure. No matter how complex the data are, they may be naturally represented as *data graphs* in which data are stored on edges and nodes are object identities to glue those edges [1,4].

Data graphs as well as query languages over graphs have been studied over two and half decades [24]. Almost graph queries are based on regular expressions defined over the alphabet of edge/node labels [5, 7, 9, 13], for example, *conjunctive queries*, *regular path queries*, or *conjunctive regular path queries*. Besides, graph transformations have also been studied, and it plays an important role in model transformations [6, 11].

Recently, with the explosion of Big Data, graphs become bigger and bigger. However, the sequential graph processing library for UnQL (a well-known SQL-like graph query language) can only deal with graphs of maximum ten thousand nodes with good scalability [5]. The scalability of GraphQL [10] is also deemed as an open problem. Hence, it is necessary to reconsider queries as well as transformations for big graphs.

To tackle the problem of scalable processing, Google proposed a novel model, named *Pregel*, to process big graphs in a distributed way [15]. It was inspired by the *Bulk-Synchronous Parallel* (BSP) model [21] whose computation consists of

Le-Duc Tung, Dept. of Informatics, SOKENDAI (The Graduate University for Advanced Studies), Japan.
Chong Li, Programming Research Laboratory, National Institute of Informatics (NII), Japan.
Xiaodong Meng, Shanghai Jiao Tong University, China.
Zhenjiang Hu, Programming Research Laboratory, National Institute of Informatics (NII), Japan.

a sequence of *supersteps*. The BSP model makes reasoning about programs easier. Besides, during each superstep, a common function is applied to *each vertex* instead of a subgraph leads to a fault-tolerant execution, which is very important to big graph processing.

However, to our knowledge, there exists few frameworks on top of Pregel that allow executing graph queries and transformations. In [16], Nolé et al. have tried to port a part of GXPath [13] to Pregel. Krause et al. [12] have implemented a subset of the transformation unit types supported by the Henshin [2] model transformation tool.

This paper aims to propose a framework on top of Pregel to allow graph queries and transformations. We borrow UnQL [5], an SQL-like language over graphs, as the interface language. We propose a systematic approach to automatically compile UnQL queries into distributed Pregel programs. It is worth to note that UnQL queries return data as graphs and their syntax allows us to do both graph queries and transformations.

Our contributions in this paper are as follows.

- We defined a domain-specific language (DSL) based on structural recursion over graphs. This DSL is a subset of UnCAL – the internal algebra of UnQL – and it allows us to express "vertical" graph computation along a path of edge labels with branch conditions.
- We parallelized the DSL to obtain efficient and scalable Pregel programs. This procedure are automatically done by our framework and give us space to do further optimizations.
- We defined rules to translate UnQL queries to the DSL. We have sucessfully dealt with basic queries such as *conjunctive queries*, *regular path expression queries* and *conjuntive regular path queries*, as well as transformations based on regular path expressions.
- We performed experiments on real data such as citation networks, Youtube videos and Amazon product datasets. Experimental results showed that our framework has a good scalability, and its performance may overtake *Spark SQL* [3] – a new relation processing for big data using HiveQL – if HiveQL queries contain a large number of *left outer joins*.

This paper is organized as follows. Section 2 introduces the data model and query language of our framework. Our approach to process UnQL queries with Pregel is proposed in Sec. 3, where we show an internal algebra and how to parallelize the algebra in Pregel. Section 4 shows an implementation of our approach in GraphX (a open source library written in Spark to process big graphs) and its experimental results. Related work is discussed in Sec. 5. Sec. 6 concludes the paper.

2 Data Model and Query Language

2.1 Data Model

Data is represented as *rooted, directed edge-labeled graphs*. In this model, data are stored on edge labels and each vertex has a unique identity. A graph has one or multiple vertices designated as roots. One could consider this model as an extension of tree-structured model. Figure 1 shows an example of a rooted, directed labeled graph representing a citation network to store papers and their citation relationships, where the black-box vertex denotes the root of the graph.

Input graphs may be stored in a single big file or in multiple separated files. Each file contains a set of edges, each edge occupies one line of file. The format for a line is

```
src_id is_root dest_id edge_label
```

where, `src_id` and `dst_id` are the ID of source and destination vertices of the representing edge, respectively. The field `is_root` indicates that if the source node is the root (value 1) or not (value 0). The field `edge_label` is the exact data (a string)

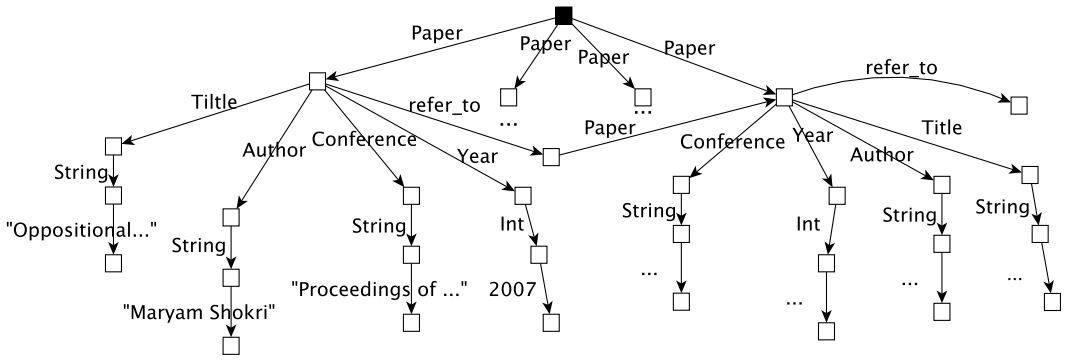


図 1 A Rooted, Directed Edge-Labeled Graph of Paper Citation Network.

we want to store on the edge.

2. 2 Query Language

The interface language of our framework is borrow from *UnQL* [5], an SQL-like language over graphs. A query has a `SELECT...WHERE...` surface.

$Query ::= \texttt{SELECT}\ G\ \texttt{WHERE}\ BC_1, \ldots, BC_n$

The `SELECT` part is a graph called by variable, or constructed either by *graph constructors*, or by *structural recursion functions*.

$$
\begin{array}{rcl}
G & ::= & \$g \\
& | & \{\} \\
& | & \{L : G\} \\
& | & G_1 \cup G_2 \\
& | & \{G_1, \ldots, G_n\} \\
& | & \textbf{letrec sfun} \\
& & \quad f(\{L : \$g\}) = G_1 \\
& & \quad \ldots \\
& & \textbf{in}\ f(G_1) \\
& | & (Query)
\end{array}
$$

Variables are identifiable by their starting symbol \$. $\{\}$ constructs a root-only graph, $\{a : G\}$ constructs a graph by adding an edge with label L pointing to the root of graph G, $G_1 \cup G_2$ unions two graphs into one by adding two ϵ-edges from the new root to the roots of G_1 and G_2, and $\{G_1, \ldots, G_n\}$ unions all graphs $G_1, \ldots, G_n$ into one by using n ϵ-edges. **letrec sfun** is used to define structural recursion functions, and the **in** after is used to call one defined function. More detail about definition of structural recursion function is shown in Section 3. 2. Query may be nested.

The `WHERE` part is a list of *bind* conditions and *boolean* conditions for pattern matching. A *boolean* condition is to compare if a label variable is equal to a given value. A *bind* condition is used to match a graph with a given pattern. A pattern is a graph constructed by graph constructors where label L may be a *regular pattern path* (RPP). A RPP is a sequence of labels or wildcard "_".

$$
\begin{array}{rcl}
\text{BC} & ::= & \text{BindCond} \mid \text{BoolCond} \\
\text{BindCond} & ::= & \text{Pattern IN G} \\
\text{Pattern} & ::= & \{\text{RPP:Pattern}\} \mid \text{G} \\
\text{RPP} & ::= & \text{L} \mid \_ \mid \text{RPP.RPP} \mid \text{RPP*}
\end{array}
$$

2. 3 Examples

The simplest query is a regular path query. Assume we want to return all papers' title from a citation graph. The result is a graph with one root and its outgoing edges are paper titles. Such query is written as follows.

```
Q1 = SELECT $t
     WHERE {Paper.Title.String:$t} IN $db
```

In UnQL queries, there always has a special variable `$db` refering to the input graph. In the query

Q1, the variable $t is bound to each graph that follows by a path *starting from a root* in which the concatenation of its edge labels satisfies the regular expression Paper.Title.String. The result graph is a union of graphs bound by $t.

We can reorganize returned data to construct a new graph.

```
Q2 = SELECT {Article: ({Topic:$t} U $y)}
     WHERE {Paper:$p} IN $db,
           {Title.String:$t} IN $P,
           {Year.Int:$y} IN $p
```

In Q2, we first bind variables $t and $y to the title and the year of paper $p, respectively. After that, it constructs, for each paper, a graph that has one edge labeled Article pointing to a union of two graphs: one has one edge labeled Topic pointing to the graph $t, and the other is the graph $y.

We can define a conjunctive regular path query by using conditions over edge labels as follows.

```
Q3 = SELECT $p
     WHERE {Paper:$p} IN $db,
           {Year.Int:{$y:{}}} IN $p,
           $y = 2015
```

The query Q3 combines regular path expressions and conditions over edge labels. This query returns papers written in 2015.

We can also perform transformations over graphs. For example, for each paper returned by the query Q3, we relabel edges Conference to Venue.

```
Q4 = SELECT
     letrec sfun
       c2v({Conference:$g}) = {Venue:c2v($g)}
       c2v({$l:$g}) = {$l:c2v($g)}
     in c2v($p)
     WHERE {Paper:$p} IN $db,
           {Year.Int:{$y:{}}} IN $p,
           $y = 2015
```

Here, we use a structural recursive function *c2v* to relabel edges Conference. The function *c2v* is defined with matching patterns. This function does not change the structure of the input graph but only modify the label of edges from Conference to Venue.

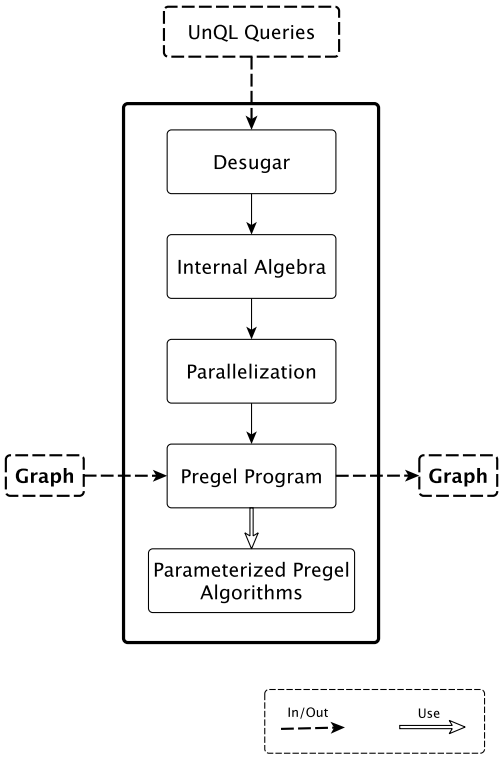


図 2 Overview of our Framework

3 Processing UnQL Queries

3.1 Overview of Our Framework

Figure 2 shows the input, the output and the major components of our framework. Details are described in the following.

- The **Desugar** accepts an input UnQL query, parses it and generates a specification following the syntax of our Internal Algebra.
- The **Internal Algebra** is based on structural recursion over graphs, and is a subset of *UnCAL* (an internal algebra of UnQL). There is the absence of label variable comparisons, say, $\$l_1 = \$l_2$, where $\$l_1$ and $\$l_2$ are two label variables belonging to two different recursive func-

tions. In that sense, our internal algebra uses the pure structural recursion over graphs.

- The **Parallelization** generates Pregel programs from specifications written in the Internal Algebra. Depending on different specifications, it will generate different evaluation strategies. After that, it compiles such strategies into a Pregel program that utilizes parametrized Pregel algorithms. Optimization rules such as Fusion, Tupling are implemented in this component.
- The **Parametrized Pregel Algorithms** consists of efficiently-made functions that are used to evaluate strategies generated by the Parallelization. These functions are Pregel algorithms.
- The **Pregel Program** is the main program that is generated by the Parallelization component. Users run this program with an input graph and the result is a new graph stored on files having a format mentioned in Sec. 2. 1.

3. 2 Internal algebra

Figure 3 shows the syntax of our language. A program starts with a header that specifies a composition of functions followed by a sequence of function declarations. Declarations are defined in the way of pattern matching and its body is an expression. For a function f, its argument is in the form of $\{l : \$g\}$ that is one of graph constructors presenting a graph constructed by appending the edge labeled l to the root of the graph $\$g$. Note that, l can be a real label a or a label variable $\$l$. Declarations of f are based on pattern matching for $\{l : \$g\}$. Only one $f(\{\$l : \$g\})$ is allowed and must be located after all other declarations of $f(\{a : \$g\})$. The declaration $f(\{\$l : \$g\})$ will apply for graphs that do not match previous patterns.

The body of a declaration is an expression including nine graph constructors, graph variables, function applications and **if_then_else** conditions. We require a strict form for function applications in which only one graph variable is allowed as its argument, which avoids computations that may lead to infinite loop. Due to the limitation of space, we ignore the details of graph constructors. Readers may refer to [5] for more information.

The semantics of our language is as follows. Given a set of structural recursive functions (defined by declarations), and a rooted edge-labeled graph, the program returns a new rooted edge-labeled graph by applying a transformation defined by the composition of structural recursive functions. Function composition is denoted by "$\circ$", and, from its definition, we have $(f_2 \circ f_1)\, x = f_2\,(f_1\, x)$. A declaration $f(\{l : \$g\})$ means, for each edge labeled l and its following subgraph $\$g$ in the input graph, we do some computations on l and then apply the structural recursive functions f on $\$g$. Results returned by applying a function f on adjacent edges are automatically combined by the constructor $\cup$ as follows: $f(G_1 \cup G_2) = f(G_1) \cup f(G_2)$.

3. 3 Rules to transform query to the Internal algebra

Translating `SELECT...WHERE` queries into our internal algebra consists of several steps. We need to first desugar `WHERE` clause of queries in order to have a list of uniformed bind conditions. After that we can generate a dependency tree of graph variables and translate bind conditions into structural recursion functions one by one recursively. At the end, query on internal algebra is created by using generated structural recursion functions.

3. 3. 1 Desugaring query

Conditions of `WHERE` clause of a query must be reformatted to the form

$$\{\text{Rpp} : \text{Var}\}\ \texttt{IN}\ \text{Var}$$

before transforming into our Internal algebra.

$$
\begin{array}{rcll}
prog & ::= & \textbf{main}\ f\,[\circ\ f]\ \textbf{where}\ decl \cdots decl & \{\ \text{program}\ \} \\
decl & ::= & f(\{l : \$g\}) = t & \{\ \text{structural recursive function}\ \} \\
t & ::= & \{\} \mid \{l : t\} \mid t \cup t \mid \&x := t \mid \&y \mid () & \{\ \text{graph constructors}\ \} \\
 & \mid & t \oplus t \mid t \mathbin{@} t \mid \textbf{cycle}(t) & \{\ \text{graph constructors}\ \} \\
 & \mid & \$g & \{\ \text{graph variable}\ \} \\
 & \mid & f(\$g) & \{\ \text{function application}\ \} \\
 & \mid & \textbf{if}\ bcond\ \textbf{then}\ t\ \textbf{else}\ t & \{\ \text{if\_then\_else}\ \} \\
bcond & ::= & \textbf{isempty}(t) & \{\text{an expression returns an empty graph not}\} \\
 & \mid & bcond\ \&\&\ bcond & \{\ \text{AND condition}\ \} \\
 & \mid & bcond \mid\mid bcond & \{\ \text{OR condition}\ \} \\
 & \mid & !bcond & \{\ \text{NOT condition}\ \} \\
l & ::= & a \mid \$l & \{\ \text{label}\ (a \in String)\ \text{and label variables}\ \}
\end{array}
$$

図 3 Syntax of our DSL Language

Rule 1.

```
case  WHERE {Rpp_1:Pat_1,
               ..., Rpp_n:Pat_n} IN Var
to    WHERE {Rpp_1:Pat_1} IN Var,
            ...,
            {Rpp_n:Pat_n} IN Var
```

This rule splits one bind condition into several bind conditions if a bind condition includes more than one pattern.

Rule 2.

```
case  WHERE {Rpp_1:Pat_1,
               ..., Rpp_n:Pat_n} IN G
to    WHERE G IN NewVar,
            {Rpp_1:Pat_1} IN NewVar,
            ...,
            {Rpp_n:Pat_n} IN NewVar
```

This rule is similar to Rule 1. When the graph G to match is not a graph variable but more complex one, then we create a graph variable to handle this graph before pattern matching.

Rule 3.

```
case  WHERE {Rpp:Pat} IN Var
to    WHERE {Rpp:NewVar} IN Var,
            Pat IN NewVar
```

This rule is used to flatten encapsulated patterns into a list of singletons.

Rule 4.

```
case  WHERE {} IN Var
to    WHERE isempty(Var)
```

For matching an empty graph that has not label, we need to use `isempty` command.

3.3.2 Constructing dependency tree

After desugaring where clause, we construe a dependency tree of graph variables that are used by the conditions. The root of the dependency tree is the variable `$db`, which is also the root of input graph. The children nodes of the root are the suggraph variables defined by the conditions that call `$db`. Recursively, the children nodes of each of these nodes are the suggraph variables defined by the conditions that call the variable of current node. If a condition is applied on a graph variable that was not defined by other binding condition, then it will be dropped automatically.

3.3.3 Translating into structural recursion

In this step, each binding condition is translated into two structural recursion functions, where the body of the functions is based on the condition, and the functions are applied to the graph that is bound to the condition. One of these structural recursion functions, called *fv*, is used to return re-

sult graph, the other function, called *fc*, is used to check if other conditions that are applied on subgraphs are all satisfied. We here also generate a function, called *fr*, based on `SELECT` part to construct queried graph using current graph and the constructed graphs from the children nodes.

Rule A. For a bind condition {`Label : SVar`} `IN GVar`, where `SVar` is a leaf in the dependency tree and it is bound by a boolean condition **isempty**(SVar), this bind condition is translated to

fc`(Label:SVar) =` **if** `(isempty(SVar))`
then {`SATISFIED:{}`} **else** {}
fc`(AnyVar:SVar) =` {}
fv`(Label:SVar) =` **if** *fc*`(Label:SVar)`
then *fr*`(SVar,{})` **else** {}
fv`(AnyVar:SVar) =` {}
with function call *fv*(`GVar`).

Rule B. For a bind condition {`Label : SVar`} `IN GVar`, where `SVar` is a leaf in the dependency tree it is not bound by any condition, this bind condition is translated to

fc`(Label:SVar) =` {`SATISFIED:{}`}
fc`(AnyVar:SVar) =` {}
fv`(Label:SVar) =` *fr*`(SVar,{})`
fv`(AnyVar:SVar) =` {}
with function call *fv*(`GVar`).

Rule C. For a bind condition {`Label : SVar`} `IN GVar`, where `SVar` is a node in the dependency tree, and *fv1*, ..., *fvn*, *fc1*, ..., *fcn* are the structural recursion functions translated from the bind conditions of the children node/leaf of `SVar`, the current bind condition is translated to

fc`(Label:SVar) =` **if** `(`*fc1*`(SVar) &&`
`... &&` *fcn*`(SVar))`
then {`SATISFIED:{}`} **else** {}
fc`(AnyVar:SVar) =` {}
fv`(Label:SVar) =` **if** *fc*`(Label:SVar)`
then *fr*`(SVar,` *fv1*`(SVar) U`
`...` `U` *fvn*`(SVar))` **else** {}
fv`(AnyVar:SVar) =` {}
with function call *fv*(`GVar`).

The top level structural recursion function is called by the **main** program of our internal algebra, and the definition of all structural recursion functions are declared in the **where** clause of the internal algebra.

3.4 Parallelization of the Internal Algebra in Pregel

The key idea to parallelize the internal algebra is to transform the evaluation of the internal algebra to an efficient algorithm in Pregel. Here, efficient algorithms refer to the ones satisfying the constraints for *Practical Pregel Algorithms* in [26]. A Pregel algorithm might consist of one or many supersteps. Sometimes we call it a Pregel phase.

We classify specifications written in the internal algebra into two classes: one with **if_then_else** conditions and the other without **if_then_else** conditions. Our approach is firstly finding an efficient solution for specifications without **if_then_else** conditions, and then using that solution to build up an efficient solution for specifications with **if_then_else** conditions.

3.4.1 Specifications without if_then_else

An effcient solution for specifications without **if_then_else** conditions are proposed in [20], in which each of such those specifications is evaluated by three Pregel phases as follows.

$$eelim \circ bulk'_{\mathcal{F}}(e_\pi) \circ mark_{\{\&f_s\}}(e_\rightarrow)$$

where *mark* is a multi-step Pregel phase whose vertex computation is defined by $e_\rightarrow$. *bulk* is an one-step Pregel phase that applies the function e_π on each edge. *eelim* is a multi-step Pregel phase to eliminate ε-edges producing during the *bulk* computation. Basically, *eelim* computes the transitive closure of ε-edges. f_s is the function between keywords **main** and **where**, it denotes the starting point or the main function of the specification. In

the case we have a composition of functions between **main** and **where**, say $f_1 \circ f_2 \cdots \circ f_k$, f_s is f_1 in general. $\&f_s$ denotes a marker built from the function f_s. $\mathcal{F}$ is a set of functions in the sequences of function calls starting from f_s. The function $e_{\rightarrow}$ defines a transition table in which inputs include a function marker and an edge label, output is a set of function markers that will be called *in the body* of the input function. The function e_{π} accepts a function marker and an edge label, and call the appropriate pattern matching in the specification, corresponding to the function name and the edge label.

Example 1 The following query finds all graphs following by a path expression $\_*.c$, and then transforms all edges labeled `b` in those graphs into edges labeled `d`.

```
SELECT
  letrec sfun
    b2d({b:$g}) = {d:b2d($g)}
    b2d({$l:$g}) = {$l:b2d($g)}
  in {c:b2d($r)}
WHERE {_*.c:$r} IN $db
```

To evaluate this query, we firstly translate it into a specification in the internal algebra. Basically, we take care of regular path epressions and rewrite them into structural recursive funtions. Functions defined by **letrec sfun** are already in the form of structural recursion, hence we keep them unchanged. In particular, the specification of the query is the following.

```
main f1 where
  f1({c:$g}) = {c:b2d($g)} U f1($g)
  f1({$l:$g}) = f1($g)
  b2d({b:$g}) = {d:b2d($g)}
  b2d({$l:$g}) = {$l:b2d($g)}
```

From the above specification, we extract two functions $e_{\rightarrow}$ and e_{π} in order to generate a Pregel program. This step is done by Compiler.

$e_{\rightarrow} = \lambda(\&z,\ \$l).$
  $(\&z,\ \$l)$ **match** {
    case $(\&f_1, \mathtt{c}) \Rightarrow \{\&b2d\}$
    case $(\&f_1, \_) \Rightarrow \{\&f_1\}$
    case $(\&b2d, \mathtt{b}) \Rightarrow \{\&b2d\}$
    case $(\&b2d, \_) \Rightarrow \{\&b2d\}$
  }

$e_{\pi} = \lambda(\&z,\ \$l).$
  $(\&z,\ \$l)$ **match** {
    case $(\&f_1, \mathtt{c}) \Rightarrow \&f_1 := \{\mathtt{c} : \&b2d\} \cup \&f_1$
    case $(\&f_1, \_) \Rightarrow \&f_1 := \&f_1$
    case $(\&b2d, \mathtt{b}) \Rightarrow \&b2d := \{\mathtt{d} : \&b2d\}$
    case $(\&b2d, \_) \Rightarrow \&b2d := \{\$l : \&b2d\}$
  }

Figure 4 shows intermediate graphs generating during the evaluation. Firstly, a marker graph is created by the Pregel phase *mark*. The marker graph is computed as follows. First, the root vertex is initialized with a singleton set $\{\&f_1\}$, where $\&f_1$ is the marker in our program. We evaluate the first edge $(u, \mathtt{a}, v)$ from the root. Its result, $e_{\rightarrow}(\&f_1, \mathtt{a}) = \{\&f_1\}$, is written to the vertex v. Next, we concurently evaluate two edges $(v, \mathtt{b}, w_1)$ and $(v, \mathtt{c}, w_2)$ emanating from v, and results are written to respective targets w_1, w_2. This procedure is iterated and then terminated when it can not find new markers to add to vertices. A bulk graph is then computed by the *bulk* as follows. For each vertex u, and its set of markers $\mathcal{X}_u$, we create $|\mathcal{X}_u|$ disjoint vertices. Next, we apply the function e_{π} on each edge (u, l, v) and each marker in $\mathcal{X}_u$, producing a subgraph of $|\mathcal{X}_u|$ input markers. In Fig. 4, these subgraphs are surrounded by a shaded rectangle. After that, we use ε-edges to connect disjoint vertices and subgraphs. Finally, the phase *eelim* eliminates all ε-edges in the bulk graph to produce the final result. □

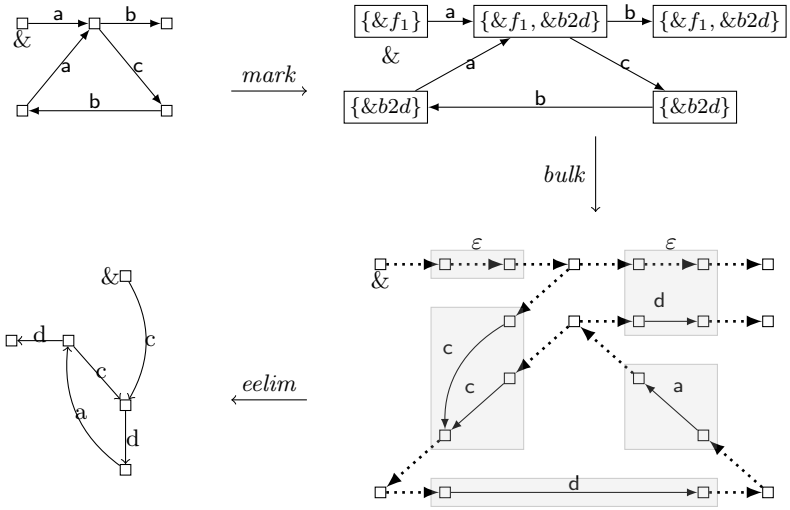


図 4 The Flow of Computation for the Example 1. Values inside a box are vertex values not vertex ids. The root node is marked by &

3. 4. 2 Specifications with if_then_else

We now turn to show how to evaluate specifications containing **if_then_else** statements. The difficulty in evaluating such specifications is relating to the computation for each edge. Recall that each declaration $f(\{l : \$g\})$ describes a computation for an edge labeled l. Once there exists a **if_then_else**, the function f certainly depends on the graph $\$g$, which is difficult to be implemented in Pregel where each vertex only knows its outgoing edges instead of the whole following graph $\$g$. Our idea is evaluating all branches **if**, **then**, **else** at the same time by a specfication without **if_then_else**, then using an iterative Pregel algorithm to check conditions in branches **if**, and finally using another specfication without **if_then_else** to extract final results from branches **then** and **else**.

We sketch our idea via a simple example. Consider the graph in Fig. 1, we write a specification to return all titles of papers published in 2010. For brevity, we ignore edges `Int` connecting edges `Year` and its value (say, edge `2007`), hence, for example, the path `Year.Int.2007` now becomes `Year.2007`.

main f_1 **where**

$$
\begin{array}{lcl}
f_1\,(\{\texttt{Paper} : \$g\}) & = & \textbf{if}\ !isemtpy(f_2(\$g)) \\
 & & \textbf{then}\ f_3(\$g) \\
 & & \textbf{else}\ \{\} \\
f_1\,(\{\$l : \$g\}) & = & \{\} \\
 & & \\
f_2\,(\{\texttt{Year} : \$g\}) & = & f_{21}(\$g) \\
f_2\,(\{\$l : \$g\}) & = & \{\} \\
 & & \\
f_{21}\,(\{\texttt{2010} : \$g\}) & = & \{\texttt{2010} : \{\}\} \\
f_{21}\,(\{\$l : \$g\}) & = & \{\} \\
 & & \\
f_3\,(\{\texttt{Title} : \$g\}) & = & id(\$g) \\
f_3\,(\{\$l : \$g\}) & = & \{\}
\end{array}
$$

Here, {} is a graph constructor for an empty graph. If the function f_2 returns an empty graph, then the expression $isemtpy(f_2(\$g))$ returns *True*, otherwise, *False*.

The first specification without **if_then_else** is achieved by flattening **if_then_else** statements and representing them by graphs whose edges are correspondent to keywords, e.g. **_if**, **_then**, **_else**, *_isempty* etc. Here, we use a special prefix "_" to distinguish keywords from users' data in a

graph. One could view these graphs as Abstract Syntax Trees (ASTs) of **if_then_else** statements.

main f_1 **where**

$$
\begin{array}{lcl}
f_1\,(\{\texttt{Paper} : \$g\}) & = & \{\texttt{\_match} : (\{\texttt{\_if} : \\
 & & \{\texttt{\_cond\_not} : \\
 & & \{\texttt{\_isempty} : f_2(\$g)\}\}\}) \\
 & & \texttt{U}\ (\{\texttt{\_then} : f_3(\$g)\}) \\
 & & \texttt{U}\ (\{\texttt{\_else} : \{\}\}) \\
f_1\,(\{\$l : \$g\}) & = & \{\} \\
\ldots & &
\end{array}
$$

Note that, for each **if_then_else** statement, we introduce an edge `_match` appending to the root of the **if_then_else** graph. The label of these edges will be changed to one of `_match_true` and `_match_false` during the iterative Pregel algorithm. The `_match_true` (`_match_false`) indicates that the expression of **if** returns a value *True* (*False*). These `_match` edges are also important to derive another specification without **if_then_else** in order to extract the final result.

The iterative Pregel algorithm evaluates **if** branches in order to update edges `_match`. Basically, it starts from edges `_cond_isempty` to check the following graphs of those edges are empty or not, then propagates results back to the `_match` edges. Afterwards, `_match` will be updated to `_match_true` or `_match_false`. Figure 5(b) shows an example result when applying this algorithm to the graph in Fig. 5(a).

Finally, we use another specification without **if_then_else** to extract the final result from the graph in Figure 5(b). This specification finds all edges `_match_true` and `_match_false`, then extracts graphs following edges `_then` (or `_else`) once it meets edges `_match_true` (or `_match_false`). This specification is independent with input queries/specifications, and is always written as follows.

main f_1 **where**

$$
\begin{array}{lcl}
f_1\,(\{\texttt{\_match\_true} : \$g\}) & = & f_{then}(\$g) \\
f_1\,(\{\texttt{\_match\_false} : \$g\}) & = & f_{else}(\$g) \\
f_1\,(\{\$l : \$g\}) & = & \{\$l : f_1(\$g)\} \\
 & & \\
f_{then}\,(\{\texttt{\_then} : \$g\}) & = & f_1(\$g) \\
f_{then}\,(\{\$l : \$g\}) & = & \{\} \\
 & & \\
f_{else}\,(\{\texttt{\_else} : \$g\}) & = & f_1(\$g) \\
f_{else}\,(\{\$l : \$g\}) & = & \{\}
\end{array}
$$

4 Implementation and Evaluation

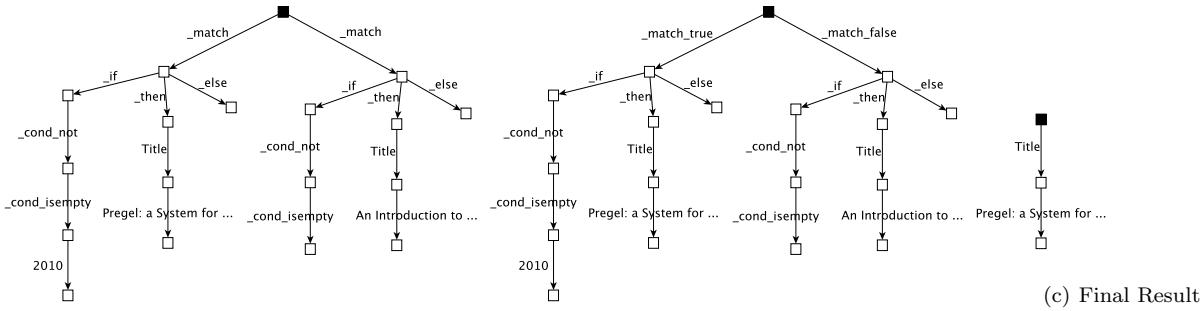
We implemented our framework over GraphX [25], an open source library for big graph processing. GraphX is a library of Spark and supports Pregel model. We used the Spark version 1.4.0 (released Jun 11, 2015).

The dataset used in our experiments is Amazon Product Co-purchasing Network†1. It contains product metadata and review information of about 548,552 different products from Amazon website.

We need firstly to convert the raw dataset of Amazon Product to a rooted directed edge-labled graph. This converted graph contains 90,227,076 vertices and 103,573,986 edges. We used the following schema (KM3 format) for our data graph.

```
package Product {
  datatype String;
  datatype Int;
  class Product {
    reference id: Int;
    reference asin: Int;
    reference title: String;
    reference group: Group;
    reference salesrank: Int;
    reference similar [0-*]: Product;
    reference category [0-*]: Category;
    reference review [0-*]: Review;
```

†1 https://snap.stanford.edu/data/amazon-meta.html



(a) First Specification without **if_then_else**

(b) Iterative Pregel Algorithm

(c) Final Result

図 5 Graphs Generated During the Evaluation of Specifications with if_then_else

```
  }
  class Category {
    reference id: Int;
    refrence name: String;
    reference contain [0-*]: Category;
  }
  class Review {
    reference time: String;
    reference customer: Customer;
    reference rating: Int;
    reference votes: Int;
    reference helpful: Int;
  }
  class Customer {
    reference id: Int;
  }
  class Group {
    reference name: String;
  }
}
```

The schema is also used by users to write UnQL queries.

Although, in general, our framework is for unstructured data presented by graphs, we compared our framework with a relational data processing framework, Spark SQL [3], to see the performance of our framework when dealing with structured data. To create an input for Spark SQL, we stored the Amazon Product dataset in Tables, where each table corresponds to one class in the KM3 schema file (listed above) and we also add more tables to make relationships between tables.

We used three UnQL Queries in our experiments in which one is regular path queries and others are conjuntive regular path queries. The first query returns the category name and the title of every product.

```
Q1 (unql) =
SELECT
   {product:
       ((SELECT {category:$c}
         WHERE
           {category.Category.name:$c} IN $p)
       U
       (SELECT {title:$t}
        WHERE  {title:$t} IN $p))
   }
WHERE {Product:$p} IN $db
```

The second query returns all information of products which belongs to group `Book`.

```
Q2 (unql) =
SELECT
  (SELECT
   {product:($a U $t U $sr U $c
```

```
                U $r U $g U $s)}
   WHERE
   {asin:$a} IN $p,
   {title:$t} IN $p,
   {salesrank:$sr} IN $p,
   {category.Category.name:$c} IN $p,
   {review.Review.customer:$r} IN $p,
   {group:$g} IN $p,
   {similar.Product.asin:$s} IN $p
 )
WHERE
  {Product:$p} IN $db,
  {group.Group.name:{$b:{}}} IN $p,
  $b = "Book"
```

The third query returns all information of products that belongs to group `Book`, in which we don't return the field `asin` of *similar products of those products*, but the field `asin` of *similar products of similar products of those products*.

```
Q3 (unql) =
SELECT
  (SELECT
   {product:($a U $t U $sr U $c
                U $r U $g U $s)}
   WHERE
   {asin:$a} IN $p,
   {title:$t} IN $p,
   {salesrank:$sr} IN $p,
   {category.Category.name:$c} IN $p,
   {review.Review.customer:$r} IN $p,
   {group:$g} IN $p,
   {similar.Product.similar.Product.asin:$s}
          IN $p
 )
WHERE
  {Product:$p} IN $db,
  {group.Group.name:{$b:{}}} IN $p,
  $b = "Book"
```

Spark SQL queries corresponds to the above UnQL queries are as follows.

```
Q1 (sql) =
SELECT product.title, category.name
FROM product
  LEFT OUTER JOIN prodcat
     ON (product.id = prodcat.productId)
  LEFT OUTER JOIN category
     ON (category.id = prodcat.categoryId)
Q2 (sql) =
SELECT product.asin, product.title,
       product.salesrank, category.name,
       customer.id, prodgroup.name,
       prod1.asin
FROM product
  LEFT OUTER JOIN prodcat
     ON (product.id = prodcat.productId)
  LEFT OUTER JOIN category
     ON (prodcat.categoryId = category.id)
  LEFT OUTER JOIN review
     ON (review.productId = product.id)
  LEFT OUTER JOIN customer
     ON (review.customer = customer.id)
  LEFT OUTER JOIN prodsimilar
     ON (product.id = prodsimilar.productId)
  LEFT OUTER JOIN product as prod1
     ON (prodsimilar.productAsin = prod1.asin)
  LEFT OUTER JOIN prodgroup
     ON (product.groupid = prodgroup.id)
WHERE prodgroup.name = "Book"
Q3 (sql) =
SELECT product.asin, product.title,
       product.salesrank, category.name,
       customer.id, prodgroup.name,
       prod2.asin
FROM product
  LEFT OUTER JOIN prodcat
     ON (product.id = prodcat.productId)
  LEFT OUTER JOIN category
     ON (prodcat.categoryId = category.id)
  LEFT OUTER JOIN review
     ON (review.productId = product.id)
```

```
  LEFT OUTER JOIN customer
     ON (review.customer = customer.id)
  LEFT OUTER JOIN prodsimilar
     ON (product.id = prodsimilar.productId)
  LEFT OUTER JOIN product as prod1
     ON (prodsimilar.productAsin = prod1.asin)
  LEFT OUTER JOIN prodsimilar as prodsim1
     ON (prod1.id = prodsim1.productId)
  LEFT OUTER JOIN product as prod2
     ON (prodsim1.productAsin = prod2.asin)
  LEFT OUTER JOIN prodgroup
     ON (product.groupid = prodgroup.id)
WHERE prodgroup.name = "Book"
```

Table 1 shows execution time in our experiments. We vary the number of workers (processors) to evaluate the performance of our framework. It is clear that our framework has a very good performance when we double the number of processors from 16 to 32. It gains a linear speedup. However, when we increase the number of processors up to 64, we could not gain linear speedup, but the execution time still decreases. This phenomenon has also been observed in other Pregel-based frameworks [8, 14]. Queries Q2 and Q3 were almost 3 times slower than Q1. This is because Q2 and Q3 are conjunctive regular path queries, and they are rewritten using three computations: 2 specifications without **if_then_else** and one iterative Pregel algorithm.

Our framework is slower than Spark SQL for simple queries, but faster than Spark SQL for complex queries that contains many joins. Looking at the table 1, we see that, for Q1, Spark SQL is much faster than our framework. This is because our framework needs to consider the whole graph, while Spark SQL just refers to two tables to obtain results. However, when we increased the number of joins, say Q2 query, two frameworks are quite close in performance. Finally, for Q3, our framework outperforms Spark SQL though we just add two more joins in the query.

5 Related Work

Graph Processing: Systematically developing graph algorithms is non-trivial due to the existence of cycles. Some works have tried to reduce problems on graphs to the ones on trees whose systematic solutions have been known. Wang et al. [22] proposed a systematic approach for graph problems via tree decomposition. Wei [23] used tree decomposition as an indexing method for answering reachability queries. Another approach is to develop a new calculus for graphs. UnCAL algebras is based on structural recursion [5]. GraphQL [10] is a graph query language whose core is a graph algebra. Compositions of graph structures are allowed by extending the notion of formal languages from strings to the graph domain. Graph grammars have been used for graph transformations in various domains [17]. However, both UnCAL and GraphQL are different in that their focus has been on graph databases. Our DSL is a subset of UnCAL language.

High-Level Framework: One of the few works on processing queries using Pregel is proposed by Nole et al. [16], in which Brzozowski's derivation of regular expressions are exploited. In consequence, queries are limited to regular path queries. Krause et al. [12] proposes a high-level graph transformation framework on top of BSP model. In particular, they implemented the framework in Giraph, an open-source implementation of Pregel model. The framework is based on graph grammars. Another approach is done by Salihoglu et al. [18], in which they have found a set of high-level primitives that capture commonly appearing operators in large-graph computations. These primitives are also implemented in GraphX library.

表 1 **Experimental Result on Amazon Product Dataset**

Workers[†2]	Q1 (unql)	Q2 (unql)	Q3 (unql)	Q1 (sql)	Q2 (sql)	Q3 (sql)
16	144	377	421	14	115	295
32	62	189	217	10	111	283
64	50	173	199	15	112	278

6 Conclusion

We proposed a framework on top of Pregel to query and transform big graphs. It supports both regular path queries and conjunctive regular path queries, as well as transformations based on them. Experimental results show that our framework gains a good performance. We also compared our framework with a relational data processing (Spark SQL) to evaluate its performance when dealing with structured data. We observed that our framework is faster than Spark SQL for complex queries that include many joins. This shows the advantage of graph databases over relational databases.

In the future, we will extend our framework to support *groupby* queries and *join* queries. For such queries, its specifications in the internal algebra need to be able to join graphs based on two edge variables that are parameters of two different structural recursive functions. It is not clear how to transform such specifications to existing specifications supported so far.

参 考 文 献

[1] Abiteboul, S., Buneman, P., and Suciu, D.: *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[2] Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G.: Henshin: Advanced Concepts and Tools for In-place EMF Model Transformations, *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*, MODELS'10, Berlin, Heidelberg, Springer-Verlag, 2010, pp. 121–135.

[3] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M.: Spark SQL: Relational Data Processing in Spark, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, New York, NY, USA, ACM, 2015, pp. 1383–1394.

[4] Buneman, P.: Semistructured Data, *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '97, New York, NY, USA, ACM, 1997, pp. 117–121.

[5] Buneman, P., Fernandez, M., and Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion, *The VLDB Journal*, Vol. 9, No. 1(2000), pp. 76.

[6] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G.: *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] Fernández, M., Florescu, D., Levy, A., and Suciu, D.: Declarative specification of Web sites with Strudel, *The VLDB Journal*, Vol. 9, No. 1(2000), pp. 38–55.

[8] Han, M., Daudjee, K., Ammar, K., Özsu, M. T., Wang, X., and Jin, T.: An Experimental Comparison of Pregel-like Graph Processing Systems, *Proc. VLDB Endow.*, Vol. 7, No. 12(2014), pp. 1047–1058.

[9] Harris, S. and Seaborne, A.: SPARQL 1.1 Query Language, March 2013. W3C Recommendation.

[10] He, H. and Singh, A. K.: Graphs-at-a-time: Query Language and Access Methods for Graph Databases, *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, 2008.

[11] Hidaka, S., Hu, Z., Kato, H., and Nakano, K.: Towards a Compositional Approach to Model Transformation for Software Development, *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, New York, NY, USA, ACM, 2009, pp. 468–475.

[12] Krause, C., Tichy, M., and Giese, H.: Implementing Graph Transformations in the Bulk Synchronous Parallel Model, *Fundamental Approaches to Software Engineering*, Gnesi, S. and Rensink, A.(eds.), Lecture Notes in Computer Science, Vol. 8411, Springer Berlin Heidelberg, 2014,

pp. 325–339.

[13] Libkin, L., Martens, W., and Vrgoč, D.: Querying Graph Databases with XPath, *Proceedings of the 16th International Conference on Database Theory*, ICDT '13, New York, NY, USA, ACM, 2013, pp. 129–140.

[14] Lu, Y., Cheng, J., Yan, D., and Wu, H.: Large-scale Distributed Graph Computing Systems: An Experimental Evaluation, *Proc. VLDB Endow.*, Vol. 8, No. 3(2014), pp. 281–292.

[15] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G.: Pregel: A System for Large-scale Graph Processing, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010.

[16] Nolé, M. and Sartiani, C.: Processing Regular Path Queries on Giraph, *EDBT/ICDT Workshops*, 2014.

[17] Rozenberg, G.(ed.): *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[18] Salihoglu, S. and Widom, J.: HelP: High-level Primitives For Large-Scale Graph Processing, *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, 2014, pp. 3:1–3:6.

[19] Statista: Statistics and facts about Social Networks, August 2015. `http://www.statista.com/topics/1164/social-networks/`.

[20] Tung, L.-D. and Hu, Z.: Towards Systematic Parallelization of Graph Transformations over Pregel, *Proceedings of the 8th International Symposium on High-level Parallel Programming and Applications*, HLPP'15, 2015.

[21] Valiant, L. G.: A Bridging Model for Parallel Computation, *Commun. ACM*, Vol. 33, No. 8(1990), pp. 103–111.

[22] Wang, Q., Chen, M., Liu, Y., and Hu, Z.: Towards Systematic Parallel Programming of Graph Problems via Tree Decomposition and Tree Parallelism, *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, 2013, pp. 25–36.

[23] Wei, F.: Efficient Graph Reachability Query Answering Using Tree Decomposition, *Proceedings of the 4th International Conference on Reachability Problems*, RP'10, 2010, pp. 183–197.

[24] Wood, P. T.: Query languages for graph databases, *ACM SIGMOD Record*, Vol. 41, No. 1(2012), pp. 50.

[25] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I.: GraphX: A Resilient Distributed Graph System on Spark, *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, 2013, pp. 2:1–2:6.

[26] Yan, D., Cheng, J., Xing, K., Lu, Y., Ng, W., and Bu, Y.: Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees, *Proc. VLDB Endow.*, Vol. 7, No. 14(2014), pp. 1821–1832.