Regular Paper

Let High-level Graph Queries Be Parallel Efficient: An Approach Over Structural Recursion On Pregel

Chong Li[1,2,a)] Le-Duc Tung[3,4,b)] Xiaodong Meng[5,c)] Zhenjiang Hu[1,3,d)]

Received: January 28, 2016, Accepted: May 16, 2016

Abstract: Graphs play an important role today in managing big data. Supporting declarative graph queries is one of the most crucial parts for efficiently manipulating graph databases. Structural recursion has been studied for graph querying and graph transformations. However, most of the previous studies about graph structural recursion do not exploit in practical the power of parallel computing. The bulk semantics, which is used for parallel evaluation of structural recursion, still impose many constraints that limit the performance of querying in parallel. In this paper, we propose a framework that systematically generates structural recursive functions from high-level declarative graph queries, then the generated functions are evaluated efficiently on our framework on top of the Pregel model. Therefore, the complexity in developing efficient structural recursive functions is relaxed by our solution.

Keywords: graph querying, parallel programming, structural recursion and high-level language

1. Introduction

Data become more and more complex today. Social networks such as Facebook, Twitter, and LinkedIn now have billions of active users [12], 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], [3].

Many distributed graph processing models and platforms were proposed in the recent years. Pregel [11] is one of the models inspired by the *Bulk-Synchronous Parallel* (BSP) model [15] whose computation consists of a sequence of *supersteps*. It follows the vertex-centric approach where a common function is applied to *each vertex*. GraphX [16] is built on the Spark cluster computing system [*1] for graphs and graph-parallel computation. It supports Pregel API. PowerGraph [5] is another distributed platform that uses another model named GAS. This model describes a step of computation in three phases: *gather*, receiving information about adjacent vertices; *apply*, taking computation from the gather phase and executing on the vertex whose neighbours we gathered; and *scatter*, updating data on adjacent edges. However, none of the existing distributed graph processing platforms provides a systematic approach to guide developer design efficient parallel algorithms. One still needs to study and develop algorithms case by case.

High-level query languages are essential for manipulating graph databases where the optimization of the processing evaluation could be done systematically by the platform. Cypher [*2] is a declarative graph query language for the graph database Neo4j that enables ad-hoc and programmatic (SQL-like) access to the graph. However, it does not scale up for very large graph datasets since its graphs are not distributed.

Recursion is widely used by functional programming language for traversing dataset, since it provides great flexibility for further optimizations. Peter Buneman et al. [4] proposed a solution, base graph structural recursion, using select-where queries with regular path patterns to query graphs. Their framework "is adequate for small input graphs with at most 1000 nodes and 10 000 edges". Structural recursion on data graphs was also studied for graph transformation [8]. Sequential evaluation of graph structural recursion has also been improved [7]. Yet, none of the previous studies practically focused on efficiently supporting large-scale graphs on a distributed environment.

Recently, we have proposed a solution [14] based on the Pregel model [11] for evaluating structural recursion on large graphs. However, structural recursive functions were still written by hand, and an expert designer of structural recursion for guarantee the efficiency of hand-written structural recursive functions is though essential. Supporting a high-level declarative language like SQL, that efficient structural recursion functions can be systematically generated from, is highly desired.

In this paper, we extend our solution to a framework that takes

1 National Institute of Informatics
2 Huawei Technologies France Research Center
3 SOKENDAI (The Graduate University for Advanced Studies)
4 IBM Research - Tokyo
5 Shanghai Jiao Tong University
a) research@chong.li
b) tung@nii.ac.jp
c) mengxiaodong1985@sjtu.edu.cn
d) hu@nii.ac.jp

[*1] Apache Spark: http://spark.apache.org/
[*2] Cypher Query Language: http://neo4j.com/docs/stable/cypher-query-lang.html

high-level graph queries as input in order to relax the complexity of designing structural recursive functions. The gap between large graph processing platform and high-level declarative querying language is thus filled by our solution. **Figure 1** shows the overview of our framework. A high-level graph query written by an end-user is transformed systematically into our internal algebra with a set of structural recursive functions, then a Pregel program will be generated by using our parametrized Pregel algorithms to guarantee the efficiency of the querying evaluation.

Contributions in this paper are as follows.

- We identified monadic queries, a useful subclass of UnQL queries that can be translated into parallel-efficient structural recursive functions.
- We proposed an approach, using pattern trees to describe the relationship between graph variables of queries, to translating all monadic queries into structural recursive functions in a systematic way.
- We used real big datasets to validate our graph querying framework. Both correctness and scalability were experimented, and the experimental results show that our solution may outperform an existing industrial solution for complex queries.

The rest of this paper is organized as follows. Section 2 reviews our data model and the graph structural recursion. Section 3 shows how structural recursion can be evaluated efficiently in parallel. Section 4 presents how to derive parallel-efficient structural recursive functions from high-level declarative graph queries. Section 5 validates our solution using real data, and Section 6 concludes the paper.

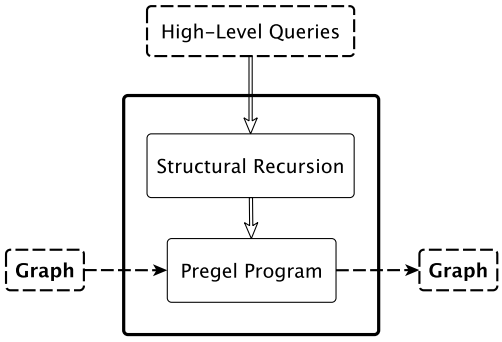


Fig. 1 Overview of our framework.

2. Data Model and Structural Recursion

2.1 Graph Data Model

Following Ref. [4], a graph is modeled as a directed and edge-labeled graph extended with *markers* and *ϵ-edges*. In this model, edges contain data, while vertices are unique identity objects without labels. Markers (with a prefix &) are symbols to designate certain vertices as *input vertices* or *output vertices*, and ϵ-edges are edges labeled with a special symbol ϵ. One could consider markers as initial/final states and ϵ-edges as "empty" transitions in automata.

Let $\mathcal{L}$ be a set of labels, $\mathcal{L}_\epsilon$ be $\mathcal{L} \cup \epsilon$, $\mathcal{M}$ be a set of markers denoted by $\&x$, $\&y$, $\&z$, ... There is a distinguished marker $\& \in \mathcal{M}$ called a *default marker*. A graph G is a quadruple $(\mathcal{V}, \mathcal{E}, \mathcal{I}, O)$, where $\mathcal{V}$ is a set of vertices, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{L}_\epsilon \times \mathcal{V}$ is a set of edges, $\mathcal{I} \subseteq \mathcal{M} \times \mathcal{V}$ 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 $v = \mathcal{I}(\&x)$ be the unique vertex such that $(\&x, v) \in \mathcal{I}$, 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. Input vertices are also called *roots* of a graph. Graphs with output vertices and ϵ-edges are only used for internal data structures for graph constructors.

Figure 2 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. For brevity, we ignore showing vertex ids.

2.2 Structural Recursion

Recursion is widely used by functional programming language for traversing dataset, since it provides great flexibility for further optimizations. However, different from list and tree structures, graph structure is much more general and complex, it can include cyclic structure. A recursion without restriction might loop infinitely on such structure. That's why we use a restricted form of recursion – structural recursion – to deal with the general structure. The restrictions of structural recursion are to ensure the termination of recursion.

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

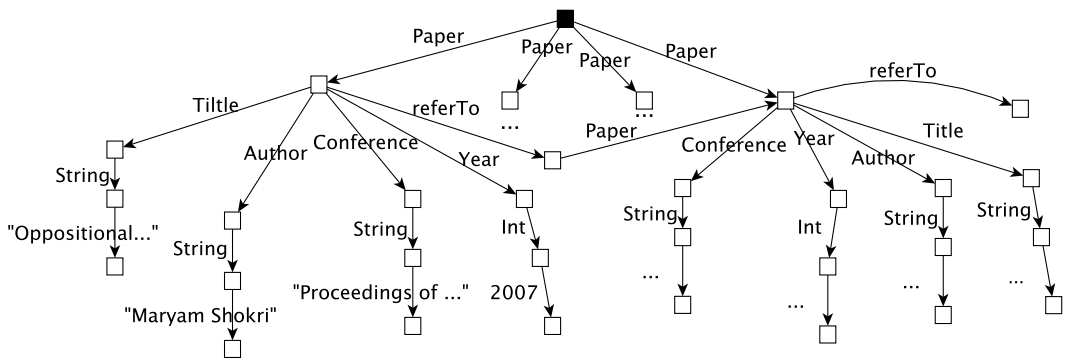


Fig. 2 A rooted, directed and edge-labeled graph of paper citation network.

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

Fig. 3 Syntax of our DSL.

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

where {} represents a root-only graph, $\{\$l : \$g\}$ represents a graph with an edge labeled by $\$l$ from root to subgraph $\$g$, $\$g_1 \cup \$g_2$ represents a graph by union of two graphs $\$g_1$ and $\$g_2$ horizontally, $\$g_1@\$g_2$ represents a graph by composing two graphs $\$g_1$ and $\$g_2$ vertically by connecting the output nodes of $\$g_1$ with the corresponding input nodes of $\$g_2$ [*3] and the expression e may contain references to variables $\$l$ and $\$g$ (but no recursive calls to f).

Since the first and the third equations are common in all structural recursions, we can describe a structural recursion in a lambda form simply defining the second equation:

$$f(\$db) = \mathbf{rec}(\lambda(\$l, \$g).e)(\$db)$$

where e could be either graph variable (denoted by $\$g$), conditional expression (**if** $l = l$ **then** e **else** e where l is either actual label or label variable $\$l$), graph constructor or structural recursion expression.

Example. Replacing all Paper-edge of the citation network by Publication-edge:

$$
\begin{aligned}
&\mathbf{rec}(\ \lambda(\$l, \$g).\quad \mathbf{if}\ \$l = \text{Paper} \\
&\qquad\qquad\quad \mathbf{then}\ \{\mathit{Publication} : \&\} \\
&\qquad\qquad\quad \mathbf{else}\ \{\$l : \&\} \qquad\qquad)
\end{aligned}
$$

3. Parallel Evaluation over Pregel

3.1 Bulk Semantics

Structural recursion can be evaluated not only recursively in a sequential way according to its definition, but also in a distributed way to benefit the computation power of parallel machine for processing big graph with a good performance. Reference [4] introduced in theory how a structural recursion could be evaluated in parallel. In a query, the matching patterns on edge-label defined in body e of structural recursion $\mathbf{rec}(\lambda(\$l, \$g).e)$ are always deterministic and countable. Therefore, we can apply the body e independently (in parallel) on every pair of $(\$l, \$g)$ on an input graph. Once all the edges were evaluated with body e, we can then reconnect the intermediate results using $\cup$ (horizontally) and @ (vertically). At the end, we clean the result graph by removing subgraphs that root cannot reach. This parallel evaluation is called *bulk semantics*.

[*3] Intuitively, input nodes are root nodes of the graph, while an output node can be seen as a "context-hole" of graphs where an input node with the same marker will be plugged later, and markers are used as an interface to connect nodes to other graphs. The default marker & is used to indicate the root.

3.2 Internal Algebra: Structural Recursive Functions

We have proposed an internal algebra for describing structural recursive functions [14]. **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 [4] 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 Parallel Evaluation over Pregel

Pregel is a model to process big graphs in a distributed way [11]. It is widely used by Google and Facebook to analyze big graphs. It was inspired by the *Bulk-Synchronous Parallel* (BSP) model [15] whose computation consists of a sequence of *supersteps*. It follows the vertex-centric approach where a common function is applied to *each vertex*. A vertex can ac-

cess its outgoing edges locally. During a superstep, a vertex receives messages from other vertices, does it computations (updating its value, mutating outgoing edges, etc.), and sends messages to other vertices. One vertex can decide not involving to the next superstep by voting to hold (inactive). A computation terminates when there is no message in transit or every vertex becomes inactive. Machines used to do vertex computations are called workers. A master is responsible for coordinating the activities of workers. A Pregel phase is a sequence of supersteps to do a computation unit, e.g., reversing a graph.

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 [18]. A Pregel algorithm might consist of one or many supersteps. Sometimes we call it a Pregel phase.

An efficient solution for specifications without **if_then_else** conditions are proposed in [14], 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.

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 labelled 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 specification without **if_then_else**, then using an iterative Pregel algorithm to check conditions in branches **if**, and finally using another specification without **if_then_else** to extract final results from branches **then** and **else**.

4. Derivation of Structural Recursive Functions from High-level Monadic Queries

4.1 Monadic Graph Queries

In this paper, we focus on a subclass of UnQL queries [4]. This subclass is so-called monadic queries, because we only treat one graph in a time. This subclass does not contain Cartesian product, GroupBy or Join, but it can simulate a large fragment of XPath 1.0 and XSLT 1.0 [9], as well as model transformations for software development [8]. For example, our monadic queries can deal with acyclic and cyclic graphs; graph transformations with new graph nodes are also supported. Syntax and notation of our monadic graph query language are borrowed from UnQL+ [*4]—an extension of UnQL to support graph editing primitives.

A query $Q(\$g)$ on a graph $\$g$ is defined as follows.

$$Q(\$g) ::= \textbf{select } C(\$g_1) \textbf{ where } P(\$g),\ P(\$g_1),\ \ldots$$

where

$$\begin{aligned} C(\$g) ::=\ & \$g \mid \{\} \mid \{a : Q(\$g)\} \\ \mid\ & Q_1(\$g) \cup Q_2(\$g) \\ \mid\ & UDF(\$g) \\ \mid\ & Q(\$g) \end{aligned}$$

and

$$\begin{aligned} P(\$g) ::=\ & \{R : \$g_1\} \textbf{ in } \$g \\ \mid\ & \textbf{isempty}(C(\$g)) \mid !P(\$g) \\ \mid\ & P(\$g)\ \&\&\ P(\$g) \mid P(\$g) \,\|\, P(\$g) \\ R ::=\ & a \mid \_ \mid R|R \mid R.R \mid R* \mid (R) \end{aligned}$$

select clause is used to define how to construct result graph using a graph variable defined in **where** clause. There are three graph constructors used in a query. The constructor {} is to construct an empty graph, $\{\_ : \_\}$ a singleton graph, and $\cup$ a union of two graphs. Queries can be nested. Besides, one can define a user-defined function (UDF) to manipulate or transform a graph. These functions are defined in the form of structural recursive functions (see Section 3.2).

where clause includes a list of predicates, which is either a binding condition or a boolean condition. A boolean condition can be either a **isempty** function that checks if a graph is empty or not. A binding condition is used to match a graph with a given pattern. A pattern is a graph composed of a regular path pattern (*RPP*) and a subgraph pointed by this *RPP*. Patterns may be bound to the same graph variable and it may also be nested. A *RPP* is a sequence of labels or *wild-cards* (_).

The binding conditions in **where** clause have two types: *retrieve* and *examine* (see Section 4.2.2). *Retrieve*-type conditions have the same semantics as UnQL; yet *examine*-type conditions do not produce as many intermediate results as the number of satisfied points in the input graph, but just one if there is at least one satisfied point or zero otherwise. This modification guarantees the efficiency and the scalability of bulk evaluation; and a result graph under our semantics is bisimilar to the result graph under UnQL's semantics.

4.2 Deriving Structural Recursive Functions

Deriving parallel-efficient structural recursive functions from a declarative graph query consists three consecutive steps: 1) desugaring the **where** clause of query, 2) generating dependency pattern tree from the desugared query, and 3) translating the pattern tree into parallel-efficient structural recursive functions.

[*4] http://www.biglab.org/demodoc/unqlplus/

Taking the following query as our running example in this section. This query is used to retrieve Title of Papers in the citation network with the condition that retrieved Papers must be published in 2007, and the subgraph that pointed by the year of publication shall be an empty graph.

q_4 = **select** {Title:\$$t$}
where {Paper:\$$p$} **in** \$$db$,
{Title:\$$t$} **in** \$$p$,
{Year.Int.2007:\$$g$} **in** \$$p$,
isempty(\$$g$)

4.2.1 Desugaring Where Clause

The first step is to desugar the **where** clause in order to standardize the patterns in the **where** clause. It lets the desugared **where** clause closer to the form of parallel-efficient structural recursive function. A desugared **where** clause is a list of binding conditions having only primitive patterns:

$\{RPP : \$g_1\}$ **in** $\$g_2$

In order to format the **where** clause, a binding condition with an union of patterns bound to the same graph shall be translated into a list of binding conditions with a single pattern bound to the same graph variable; a binding condition with nested patterns shall be flattened into a list of binding conditions with non-nested patterns; and a boolean condition shall either be plugged into binding conditions that refer to the same variable, or be dropped if its referred variable is not used by any binding condition.

At the end of this step, q_4 is transformed to

q'_4 = **select** {Title:\$$t$}
where {Paper:\$$p$} **in** \$$db$,
{Year:\$$gv_2$} **in** \$$p$,
{Int:\$$gv_1$} **in** \$$gv_2$,
{2007:{}} **in** \$$gv_1$,
{Title:\$$t$} **in** \$$p$

4.2.2 Building Pattern Tree

Let us use a tuple with three elements (rpp, pg, bg) to represent a binding condition $\{rpp : pg\}$ **in** bg. The second step is to transform the **where** clause from a list of such binding conditions to a patterns tree. This step is the key to generate structural recursive functions in our restricted form but not in a linear nested form.

A pattern tree is organized as a rose tree with the type:

$Ptree$ ::= (N(RPP, G), B($List(Ptree)$))

Each node N(rpp, g) hosts a pattern in the form $\{rpp : g\}$, i.e., a rpp and its pointing graph g. The branches B of a node is a list [*5] of $Ptree$ generated from the binding conditions that are bound to the current pattern's pointing graph. Algorithm 1 is used to transform a list of binding conditions to a set of pattern trees. The termination condition for the function `list2tree` is when the Branches set B is empty. The syntax of functions **filter** and **map** in our pseudo code are borrowed from Scala [*6], and $a.\_i$ means the i-th element of tuple a. We currently allow only one pattern binding to the root \$$db$ because of our framework. The result generated by Algorithm 1 is therefore a one-element list, and the pattern tree in this list is the final pattern tree that we need. **Figure 4** shows the pattern tree of q'_4 generated by Algorithm 1 with root \$$db$.

Algorithm 1: Transform where clause to pattern tree

Input: List of binding conditions, graph variable of root
Output: List of pattern tree
Function `list2tree`(*bcList, root*)
bcList.**filter**(_._3 = *root*)
.**map**(*bc* => (*N*(*bc*._1, *bc*._2),
B(`list2tree`(*bcList*, *bc*._2))))

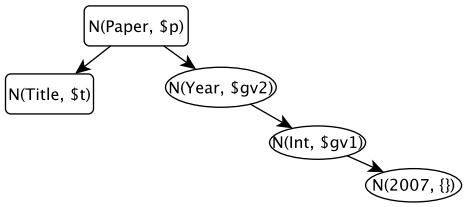


Fig. 4 Pattern tree of q'_4.

Here we add an extra information for each node: a node is either a *retrieve*-node (rectangle in Fig. 4) or an *examine*-node (ellipse in Fig. 4). A *retrieve*-node is the nodes on the path from root to the node in which its graph variable is requested by the **select** clause, otherwise it is an *examine*-node that is used to verify a condition but does not return a graph. This information is represented by a boolean for each node: true for *retrieve*-node and false for *examine*-node. If a node's pointing graph variable is the exact graph variable requested by the **select** clause, then this node is classified as *retrieve*-node, or if at least one of its branches includes a *retrieve*-node, then it is also classified as *retrieve*-node, otherwise it is classified as *examine*-node.

4.2.3 Generating Structural Recursion

At this step we use Algorithm 2 to generate structural recursive functions from a pattern tree and a graph constructor. The pattern tree was created in the previous step according to the **where** clause of a query; and the graph constructor is obtained directly from the **select** clause of the same query.

A pattern tree will be, recursively from the leaf nodes to the root node, translated into structural recursive functions. Each node of pattern tree corresponds to a structural recursive function. The function `funcgen`, that takes a pattern and a tuple with a graph constructor and an accumulator of structural recursive functions, is used to 1) generate such structural recursive function from a node, and 2) combine this newly generated function and other generated functions from the accumulator. The name of this new structural recursive function is also created by `funcgen` using letter f and an incrementing number. The *match* cases and *if* conditions in `codegen` are used to determine the situation of a node: if it has multi-branches, only one branch, or zero branch (e.g. a leaf). Here for a node has multi-branches, we also need to

[*5] The exact type should be Set, but here we use type List instead to simplify the pattern matching of B. The order in the list is meaningless.

[*6] http://www.scala-lang.org/

Algorithm 2: Generate structural recursive functions

```
   Input: Pattern tree, graph constructor from select clause
   Output: A tuple of the name of entry function and a list (accumulator)
           of structural recursive functions
   Function codegen(ptree, sg)
1      match ptree
2          case (N(rpp, pg), B(List()))
3              if N(rpp, pg) is retrieve-node then
4                  funcgen({rpp:pg}, (sg, List()))
5              else
6                  if pg is {} then
7                      funcgen({rpp:newVar},
                         ( "if isempty(newVar)
                           then {OK:{}} else {}", List()))
8                  else
9                      funcgen({rpp:pg}, ({OK:{}}, List()))
10         case (N(rpp, pg), B(bl::Nil))
11             if pg is requested graph in sg then
12                 funcgen({rpp:pg},
                       concat(List(codegen(bl, sg)),
                           (sg, List())))
13             else
14                 funcgen({rpp:pg}, codegen(bl, sg)))
15         case (N(rpp, pg), B(bl))
16             rfl := bl.filter(is retrieve-node)
                   .map(t => codegen(t,sg))
17             efl := bl.filter(is examine-node)
                   .map(t => codegen(t,sg))
18             if pg is requested graph in sg then
19                 funcgen({rpp:pg}, concat(efl, (sg, List())))
20             else
21                 if N(rpp, pg) is retrieve-node then
22                     if efl.Length > 0 then
23                         funcgen({rpp:pg},
                               concat(efl, rfl.head))
24                     else
25                         funcgen({rpp:pg}, rfl.head)
26                 else
27                     funcgen({rpp:pg},
                           concat(efl, "{OK:{}}"))
```

distinguish if it has only examine-nodes as children or it has both examine-node and retrieve-node. That is because retrieve-nodes shall return a graph and examine-nodes return only a boolean-value graph that is used to indicate whether the graph shape is satisfied (i.e. {OK:{}}) or not (i.e. {}).

For example, N(Title,t) of Fig. 4 matches the conditions in line 2 and line 3 of Algorithm 2. Therefore, we apply funcgen({Title:t}, ({Title:t}, List())), which corresponds to line 4, where the first {Title:t} is the pattern from the node, and the second {Title:t} is the graph constructor from **select** clause. As this node is a leaf in the pattern tree, the accumulator is set to empty with List(). By applying this function, we obtain:

$$f_1(\{\text{Title} : \$t\}) = \{\text{Title} : \$t\}$$
$$f_1(\{\$l : \$t\}) = \{\}$$

where entry point is f_1.

While N(2007,{}) satisfies conditions in line 2, line 5 and line 6, we thus apply line 7 and obtain:

$$f_2(\{2007 : \$gv_2\}) = \textbf{if isempty}(\$gv_2) \textbf{ then}\{\text{OK} : \{\}\} \textbf{ else } \{\}$$
$$f_2(\{\$l : \$gv_2\}) = \{\}$$

where entry point is f_2.

N(Year.Int, gv) satisfies conditions in line 10 and line 13, thus line 14 is applied. The recursive call here is for N(2007, {}) and produces f_2. A special case is that this pattern is not with a simple edge label but a regular expression path. funcgen will first transform this regular expression into an automaton, then generate structural recursive functions based on the automaton. By accumulating the function f_2 and the newly generated functions f_3 and f_4 from the automaton, we obtain:

$$f_2(\{2007 : \$gv_2\}) = \textbf{if isempty}(\$gv_2) \textbf{ then } \{\text{OK} : \{\}\} \textbf{ else } \{\}$$
$$f_2(\{\$l : \$gv_2\}) = \{\}$$
$$f_3(\{\text{Int} : \$gv\}) = f_2(\$gv)$$
$$f_3(\{\$l : \$gv\}) = \{\}$$
$$f_4(\{\text{Year} : \$gv_3\}) = f_3(\$gv_3)$$
$$f_4(\{\$l : \$gv_3\}) = \{\}$$

where entry point is f_3.

The recursion is now back to N(Paper, p). Here the conditions in line 15, line 20, line 21 and line 22, we thus apply line 23. The function concat(efl, rf) is used to generate codes that verifies if all graphs created by the functions of list efl are not empty then return rf otherwise an empty graph. Finally, we get:

eval f_5 **where**

$$f_1(\{\text{Title} : \$t\}) = \{\text{Title} : \$t\}$$
$$f_1(\{\$l : \$t\}) = \{\}$$
$$f_2(\{2007 : \$gv_2\}) = \textbf{if isempty}(\$gv_2) \textbf{ then } \{\text{OK} : \{\}\} \textbf{ else } \{\}$$
$$f_2(\{\$l : \$gv_2\}) = \{\}$$
$$f_3(\{\text{Int} : \$gv\}) = f_2(\$gv)$$
$$f_3(\{\$l : \$gv\}) = \{\}$$
$$f_4(\{\text{Year} : \$gv_3\}) = f_3(\$gv_3)$$
$$f_4(\{\$l : \$gv_3\}) = \{\}$$
$$f_5(\{\text{Paper} : \$p\}) = \textbf{if isempty}(f_3(\$p)) \textbf{ then } \{\} \textbf{ else } f_1(\$p)$$
$$f_5(\{\$l : \$p\}) = \{\}$$

4.3 Extending to Nested Queries

We allow that the graph constructor in **select** clause could be a nested query, or even union of nested queries for more complex problem. For example, a conjunctive regular path query for retrieving Title and Author of Papers published in 2007 can be written as follows:

q_5 = **select** ((**select** {Title:t}
where {Title:t} **in** p)
∪ (**select** {Author:a}
where {Author:a} **in** p))
where {Paper:p} **in** db,
{Year.Int:{$y:$g}} **in** p,
$y = 2007,
isempty($g)

For deriving the above nested query into parallel-efficient structural recursive functions, we need first derive the most-inner queries, then derive recursively outer-level queries using their entry-point function calls as graph contractor.

Query q_5 is transformed by our compiler into

eval f_6 **where**
$f_1(\{Title : \$t\}) = \{Title : \$t\}$
$f_1(\{\$l : \$t\}) = \{\}$
$f_2(\{Author : \$a\}) = \{Author : \$a\}$
$f_2(\{\$l : \$a\}) = \{\}$
$f_3(\{2007 : \$g\}) =$ **if** (**isempty**($\$g$))
then $\{OK : \{\}\}$ **else** $\{\}$
$f_3(\{\$l : \$g\}) = \{\}$
$f_4(\{Int : \$gv_1\}) = f_3(\$gv_1)$
$f_4(\{\$l : \$gv_1\}) = \{\}$
$f_5(\{Year : \$gv_2\}) = f_4(\$gv_2)$
$f_5(\{\$l : \$gv_2\}) = \{\}$
$f_6(\{Paper : \$p\}) =$ **if** (**isempty**($f_5(\$p)$))
then $\{\}$ **else** $(f_1(\$p) \cup f_2(\$p))$
$f_6(\{\$l : \$p\}) = \{\}$

where f_1 was derived from the first inner query of q_5, f_2 was derived from the second inner query, and f_3 to f_6 were from the outer query.

Queries can also be nested as a graph bound by the first binding condition. For example, we can use the following query to retrieve all publications that are referred by the papers published in 2000 by Li.

$q_6 =$ **select** {Publication:$r}
where {Refer_to.Paper:$r} **in**
(**select** $p
where {Paper:$p} **in** $db,
{Author.String.$a : $g_1} **in** $p,
$a = Li),
{Year.Int:{$y:$g_2}} **in** $p,
$y = 2000

The inner query and the outer query are both derived into the same set of structural recursion functions. The entry-point function of the inner query is composed with the one of the outer query.

Query q_6 is transformed into

eval $f_9 \circ f_4$ **where**
$f_1(\{Li : \$g\}) = \{OK : \{\}\}$
$f_1(\{\$l : \$g\}) = \{\}$
$f_2(\{String : \$g\}) = f_1(\$g)$
$f_2(\{\$l : \$g\}) = \{\}$
$f_3(\{Author : \$g\}) = f_2(\$g)$
$f_3(\{\$l : \$g\}) = \{\}$
$f_4(\{Paper : \$g\}) =$ **if** (**isempty**($f_3(\$g)$))
then $\{\}$ **else** $\$g$
$f_4(\{\$l : \$g\}) = \{\}$
$f_5(\{2000 : \$g\}) = \{OK : \{\}\}$
$f_5(\{\$l : \$g\}) = \{\}$
$f_6(\{Int : \$g\}) = f_5(\$g)$
$f_6(\{\$l : \$g\}) = \{\}$
$f_7(\{Year : \$g\}) = f_6(\$g)$
$f_7(\{\$l : \$g\}) = \{\}$
$f_8(\{Paper : \$g\}) =$ **if** (**isempty**($f_7(\$g)$))
then $\{\}$ **else** $\$g$
$f_8(\{\$l : \$g\}) = \{\}$
$f_9(\{Refer\_to : \$g\}) = f_8(\$g)$
$f_9(\{\$l : \$g\}) = \{\}$

where f_1 to f_4 were derived from the inner query of q_6, f_5 to f_9 were derived from the outer query.

5. Experiments

We implemented our solution in Spark (version 1.4.0 released on Jun 11, 2015) over GraphX [17]. We first used the Paper Citation Network dataset [13] to validate the correctness of our derivation. This dataset includes 1,632,442 papers and 2,327,450 citation relationships. The raw dataset was converted to a rooted, directed and edge-labeled graph with 6,866,730 vertices and 9,364,118 edges. We used a share-money parallel machine for the validation. The machine has two 4-core Intel Xeon E5620 2.40 GHz and 48 GB RAM.

We chose q_5 of Section 4.3 for the validation, because it is both a nested query and a conditional query. The generated structural recursive functions are the same as the ones shown in Section 4.3 for q_5, except the names of functions and of variables are different, because those were generated by our compiler using a fixed letter with an incremental number. The result graph has 334,149 vertices and 354,341 edges, and the content is exactly what we queried. **Table 1** shows execution time, speedup and efficiency of evaluation of the generated structural recursive functions from q_5.

The second dataset we used in our experiments is Amazon Product Co-purchasing Network [*7]. It includes metadata and review information of about 548,552 different products from Amazon website. The raw dataset was converted to a rooted, directed and edge-labeled graph with 90,227,076 vertices and 103,573,986 edges. The schema of the graph is shown in **Fig. 5**.

The experiments for Amazon Product Co-purchasing Network were conducted on a cluster of 16 Amazon EC2 instances of the type r3.2xlarge [*8]. Each instance has 8 processors (Intel Xeon E5-2670 v2 Ivy Bridge configured at high frequency) with 61 GB memory and using SSD as storage.

Two nested queries were prepared:

$q_7 =$ **select** {product:
(**select** {category:$c}
where {category.Category.name:$c} **in** $p)
∪
(**select** {title:$t} **where** {title:$t} **in** $p)
where {Product:$p} **in** $db

and

$q_8 =$ **select** {product:
((**select** $a **where** {asin:$a} **in** $p)

Table 1 Experimental result on citation network dataset.

Number of Processors	1	2	4	6	8
Execution Time (sec)	1506	830	460	410	339
Speepup	1	1.8	3.3	3.7	4.5
Efficiency	1	0.9	0.83	0.62	0.56

[*7] https://snap.stanford.edu/data/amazon-meta.html
[*8] https://aws.amazon.com/ec2/instance-types/

```
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; }
  class Group {
    reference name: String; }
  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; }
}
```

Fig. 5 The schema in KM3 format for the graph of Amazon Product Co-purchasing Network.

Table 2 Experimental result on Amazon Product Dataset.

# of proc	16	32	64
q_7	144	62	50
q_8	421	217	199
q'_7	14	10	15
q'_8	295	283	278

$\cup$ (**select** \$$t$ **where** {title:\$$t$} **in** \$$p$)
$\cup$ (**select** \$$sr$ **where** {salesrank:\$$sr$} **in** \$$p$)
$\cup$ (**select** \$$c$
where {category.Category.name:\$$c$} **in** \$$p$)
$\cup$ (**select** \$$r$
where {review.Review.customer:\$$r$} **in** \$$p$)
$\cup$ (**select** \$$g$ **where** {group:\$$g$} **in** \$$p$)
$\cup$ (**select** \$$s$
where {similar.Product.similar.Product.asin:\$$s$}
in \$$p$))}
where {Product:\$$p$} **in** \$$db$,
{group.Group.name:{\$$b$:{}}} **in** \$$p$,
\$$b$ = "Book"

where query q_7 is a regular path query that returns the category name and the title of every product. Query q_8 is a conjunctive regular path query that returns all information of products that belong 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.

These queries were translated into parallel-efficient structural recursive functions by our framework. The execution times (in seconds) of querying are shown in **Table 2**. We vary the number of processors to evaluate the efficiency of structural recursive functions generated by our compiler and executed on our framework. It is clear that our generated structural recursive functions have a very good efficiency 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 [6], [10]. The exact reasons of this limitation shall be studied in the future, but it is out of the scope of this paper. q_8 was almost 3 times slower than q_7. This is because q_8 is conjunctive regular path query, and it is rewritten using three computations: 2 specifications without **if_then_else** statements and one iterative Pregel algorithm.

It is not realistic to compare our flatten structural recursive functions to naively-nested structural recursions, because naively-nested ones cannot be evaluated in a distributed environment. We therefore prepared two queries, named q'_7 and q'_8 in Spark SQL [2] using Left-Outer-Join, that query the same results as q_7 and q_8 respectively, in order to 1) validate the correctness of our parallel-efficient queries generation, 2) compare the performance of our solution to an industrial solution also under Spark implementation. Our solution is slower than Spark SQL for simple queries, e.g. q_7, but faster than Spark SQL for complex queries that contains many joins, e.g., q_8. Looking at the Table 2, we see that, for q_7, Spark SQL is much faster than our solution. 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 q_8 query, our solution can outperform Spark SQL.

6. Conclusion

We have identified monadic queries and proposed a solution to systematically derive parallel-efficient structural recursive functions from these high-level declarative queries. The term of parallel-efficiency is based on the restricted form of structural recursive function and the parallel evaluation over our Pregel-based framework. We restricted the syntax of queries to guarantee the efficiency of parallel evaluation. Yet practical queries can be easily designed by a non-expert, and the performance of queries can always be insured. Using select-where queries over structural recursion to process very large graphs in a distributed environment was first studied in this paper. Experimental results show that our solution may even outperform the-state-of-the-art industrial solution when queries are complex with many joins.

In the future, we will extend supported queries to *Cartesian product*, *groupby* queries and *join* queries. However, It is not clear whether those queries can be translated to the restricted form of structural recursive function, or it is necessary to extend the restricted form of structural recursive function to support them. For such queries, the syntax and the semantics of structural recursive function need to be extended in order to be able to join graphs based on two edge variable that are parameters of two different structural recursive functions.

Acknowledgments This work was supported by JSPS Grant-in-Aid for Challenging Exploratory Research Grant Number 15K12011.

References

[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] 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, *Proc. SIGMOD'15*, pp.1383–1394 (2015).

[3] Buneman, P.: Semistructured Data, *Proc. Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp.117–121 (1997).

[4] Buneman, P., Fernandez, M. and Suciu, D.: UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion, *VLDBJ*, Vol.9, No.1, p.76 (2000).

[5] Gonzalez, J.E., Low, Y., Gu, H., Bickson, D. and Guestrin, C.: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, *OSDI*, Vol.12, No.1, p.2 (2012).

[6] 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, pp.1047–1058 (2014).

[7] Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K. and Sasano, I.: Marker-Directed Optimization of UnCAL Graph Transformations, *Proc. 21st International Conference on Logic-Based Program Synthesis and Transformation*, Berlin, Heidelberg, Springer-Verlag, pp.123–138 (2012).

[8] Hidaka, S., Hu, Z., Kato, H. and Nakano, K.: Towards a Compositional Approach to Model Transformation for Software Development, *Proc. SAC'09*, pp.468–475 (2009).

[9] Kósa, B., Benczúr, A. and Kiss, A.: Satisfiability and Containment Problem of Structural Recursions with Conditions, *Proc. ADBIS'10*, pp.336–350 (2010).

[10] 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, pp.281–292 (2014).

[11] 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, *Proc. SIGMOD'10*, pp.135–146 (2010).

[12] Statista: Statistics and Facts about Social Networks (2015), available from ⟨http://www.statista.com/topics/1164/social-networks/⟩.

[13] Tang, J., Zhang, J., Yao, L., Li, J., Zhang, L. and Su, Z.: ArnetMiner: Extraction and Mining of Academic Social Networks, *Proc. 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '08*, pp.990–998, ACM (online), DOI: 10.1145/1401890.1402008 (2008).

[14] Tung, L.-D. and Hu, Z.: Towards Systematic Parallelization of Graph Transformations over Pregel, *Proc. HLPP'15* (2015).

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

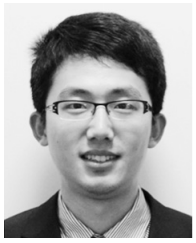
[16] Xin, R.S., Gonzalez, J.E., Franklin, M.J. and Stoica, I.: Graphx: A resilient distributed graph system on spark, *First International Workshop on Graph Data Management Experiences and Systems*, p.2, ACM (2013).

[17] Xin, R.S., Gonzalez, J.E., Franklin, M.J. and Stoica, I.: GraphX: A Resilient Distributed Graph System on Spark, *Proc. GRADES'13*, pp.1–6 (2013).

[18] 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, pp.1821–1832 (2014).

Chong Li is a senior research engineer at Distributed Algorithms Lab at France Research Center of Huawei Technologies. Before joining Huawei in 2016, he was Project Researcher at National Institute of Informatics (2014–2016), and Engineer-Researcher at EXQIM SAS (2009–2014). He also worked for many research institutes, universities and technology companies, including CNRS, INRIA, Universities of Paris and Gemalto SA. Dr. Li is qualified as Associate Professor since 2014 by French Ministry of Higher Education. He received his Doctorate degree in 2013 from Université Paris-Est, and his Engineer and Master degrees both in 2009 from 3IL and CRYPTIS, respectively. His current interest is in parallel algorithms and programming, hard real-time systems, heterogenous architectures, and their application.

Le-Duc Tung is a post-doctoral researcher at IBM Research – Tokyo, Japan. He received his PhD degree from SOKENDAI in 2016, his Master and Bechalor degrees both in computer science from HUST in 2010 and 2007, respectively. His Ph.D. thesis focused on systematic programming methods for big graphs. His interest is in algebras of programming. In particular, he studies how to use the algebras to automatically derive efficient parallel programs that deal with big data.



Xiaodong Meng is a Ph.D. student of computer science in Shanghai JiaoTong University. He received his B.S. from Wuhan University in 2007 and M.S. from Monash University in 2010. He is currently doing research in Shanghai Key Laboratory of Scalable Computing and Systems of Shanghai JiaoTong University. His main interest is in parallel and distributed computing, social graph processing and storage systems.

Zhenjiang Hu received his B.S. and M.S. degrees from Shanghai Jiao Tong University in 1988 and 1991, respectively, and Ph.D. degree from University of Tokyo in 1996. He was a lecturer (1997–2000) and an associate professor (2000–2008) in University of Tokyo, before joining National Institute of Informatics (NII) and the Graduate School for Advanced Studies (SOKENDAI) as a full professor from 2008. His main research interest is in programming languages and software engineering in general, and functional programming, parallel programming, and bidirectional model-driven software development in particular. He is a member of JSSST, IPSJ, ACM and IEEE.