Automatic Parallelization of Graph Queries with MapReduce

Tao Zan¹, Yu Liu¹ and Zhenjiang Hu²

¹ The Graduate University for Advanced Studies
{zantao, yuliu}@nii.ac.jp
² National Institute of Informatics
hu@nii.ac.jp

Abstract. The MapReduce programming model has gained attraction for large scaled data processing in recent years. One typical application is graph query parallelization. Although a lot of work has been done to parallel graph queries by MapReduce, the data model used for graphs does not reflect the original graph shape, thus making it difficult to manipulate graphs in a structured way. In this paper, we show a new approach to parallelization of graph queries in UnQL, a graph query language whose data model is an edge labelled graph. The most prominent feature of this language is that all UnQL queries can be specified by structural recursion which can be evaluated in a bulk manner. After carefully examining the properties of structural recursion, we propose a natural way for parallelizing structural recursion by MapReduce automatically in two steps: 1) bulk computing and 2) unreachable part removal and global ε-edge elimination. The experiments show the feasibility of our approach for parallelization of graph queries.

Keywords: Graph Query, UnQL, Structural Recursion, MapReduce, Parallelization

1 Introduction

Graphs are useful to represent a variety of things including the World Wide Web, social network, computer network, transportation system and many more. Since these graphs can be quite huge with millions of nodes and edges, how to parallelize graph queries to make query efficient is an important issue [8]. There is a lot of work dealing with query languages for (RDF) graphs using MapReduce. Mohammad Husain et al. [6] process queries for large RDF graphs using Hadoop and MapReduce. Kurt Rohloff et al. [10] use the Clause-Iteration approach to process SPARQL queries [9]. HyeongSik Kim et al. [7] formalize SPARQL queries on MapReduce using a Nested TripleGroup Algebra (NTGA). However, the RDF graph is stored as a collection of binary relations (triples), which do not directly reflect the original graph shape, making it hard to manipulate graph structures and express queries with paths on the graph.

In this paper, we consider parallelization of a graph query language called UnQL (unstructured query language) proposed by Peter Buleman et al. in [4].
The most prominent feature of this language is that all UnQL queries can be specified by structural recursion which can be evaluated in a bulk manner. We aim at a new way of processing graph queries by MapReduce, by parallelizing structural recursion based on its bulk semantics.

The MapReduce programming model [5] is suitable for processing large amounts of data. This programming model provides an easy way for parallel programming and has a desirable scalability that computing nodes can easily be added to the cluster.

We shall propose a new approach to parallel graph query processing, by translating UnQL query into structural recursion, then evaluating structural recursion by MapReduce. By carefully examining the properties the structural recursion, we show that a set of queries can be highly parallelized in MapReduce. More specifically, we analyze the core evaluation rules of structural recursion on this graph model, and get three decomposable properties for UnQL queries. We decompose graph queries and compute these queries by MapReduce in two steps: (1) bulk computing and local $\varepsilon$-edge elimination. (2) a new algorithm for unreachable part removal and global $\varepsilon$-edge elimination.

The remainder of this paper is organized as follows. In Section 2 we give a brief overview of the graph query language UnQL and the MapReduce programming model. In Section 3 we present techniques for graph query parallelization. In Section 4 we describe our experimental results from the use of the approach in a cloud, and conclude the paper and discuss the future work in Section 5.

2 Background

2.1 UnQL: A Graph Query Language

To understand when a query can be decomposed and how to evaluate it by MapReduce later, we start by giving a brief description of the basic concepts of this graph query language UnQL [4].

**Data Model** In UnQL, graphs are rooted, edged-labelled in the sense that all information is stored on edges while nodes have no meanings. A graph $G$ can be represented by a quadruple $(V,E,I,O)$, in which $V$ means a set of nodes, $E \subseteq V \times (\text{Label} \cup \{\varepsilon\}) \times V$ is a set of edges, $I \subseteq M \times V$ is a set of tuples of an input marker and the corresponding node, $O \subseteq V \times M$ is a set of tuples of an output marker and the corresponding node.

**Extended Graph Bisimulation** Graph bisimulation defines the value equivalence between two graphs. Intuitively, if two graphs $G_1$ and $G_2$ are bisimilar, then for every node $x_1$ in graph $G_1$ has a counterpart node $x_2$ in graph $G_2$ and vice versa. For each edge from $x_1$ to $y_1$ in graph $G_1$, there is a corresponding edge from $x_2$ to $y_2$ in graph $G_2$ and vice versa. While UnQL data model extends the graph bisimulation called *extended bisimulation* by (1) requiring edge label equality, (2) allowing adding one or more consecutive $\varepsilon$-edges to an edge(\varepsilon$-edge
can be seen as an edge with no actual meanings, just for connecting two nodes which plays an important role in bulk semantics), (3) requiring correspondence between input nodes of graph \( G_1 \) and \( G_2 \), (4) requiring correspondence of output markers of corresponding nodes. Please refer to [4] for the complete definition.

By defining \textit{extended graph bisimulation}, after eliminating \( \varepsilon \)-edges of the graph \( G_1 \) constructed by structural recursion (we will describe later), the result graph \( G_2 \) is value equivalent with graph \( G_1 \).

Core Algebra: UnCAL The internal graph algebra for the graph query language UnQL is called UnCAL (Unstructured Calculus). We depict its core syntax as follows which consists of nine graph constructors, label variables, conditionals, and the most important one called structural recursion. The core UnCAL syntax is summarized as follows:

\[
e ::= \{ \} \mid \{ \ell : e \} \mid e \cup e \mid &x := e \mid &y := e \mid \cdot \mid e \oplus e \mid e \otimes e \mid \text{cycle}(e) \mid \text{if} \ (l = l) \ \text{then} \ e \ \text{else} \ e \mid \text{rec}(\lambda (\ell, g). e)(e)
\]

\[
l ::= a \mid \ell
\]

\{ \} constructs a single node graph, \{ \ell : e \} means the graph with an edge pointing to a graph, \( \oplus \) means disjoint graph union, \( \otimes \) appends two graphs, \text{cycle} connects the output nodes and input nodes to form cycles. We will focus on structural recursion, which is a powerful mechanism in UnCAL. Prior work [4] shows that all select-where queries can be translated into structural recursion queries. This gives us a way for optimizing graph queries which we will describe later in this paper.

\textit{Structural Recursion} Structural recursion is a recursive computation function \( f \) defined as follows:

\[
f(\{\}) = \{}
\]

\[
f(\{\ell : g\}) = e(\ell, g) \otimes f(g)
\]

\[
f(g_1 \cup g_2) = f(g_1) \cup f(g_2)
\]

Here \( \ell \) is a label variable and \( g \) is a graph variable denoting the sub-graph directly pointed by the edge with label \( \ell \). Note that \( e \) may refer to both \( \ell \) and \( g \), but cannot call \( f \) recursively. Since the first and the third equations are common, we abbreviate the above definition simply as

\[
f = \text{rec}(\lambda (\ell, g). e(\ell, g))
\]

\textit{Bulk Semantics} The computation of \( f \) on graph \( G \) includes the following steps. First, making \( m \) copies of each node \( u \) in the graph \( G \) (\( m \) is the number of markers in \( e \)). Then applying to each edge with the \( e \) to get a graph with \( m \) input nodes and \( m \) output nodes. Finally, connecting the corresponding nodes with \( \varepsilon \)-edges. For a more detailed description, readers can refer to [4]. As an
example, consider the following structural recursion to extract all sub-graphs pointed by an edge labelled $a$:

$\text{getSub}(\$db) = \&z_1 @ \text{rec}(\lambda(l, g).$

\[
\begin{align*}
\text{if } l = a & \text{ then } (\&z_1 = \{\varepsilon: \&z_2\} ; \&z_2 = \{l: \&z_2\}) \hfill \\
\text{else } & (\&z_1 = \{\varepsilon: \&z_1\} ; \&z_2 = \{l: \&z_2\})(\$db)
\end{align*}
\]

As seen in Figure 1, applying structural recursion on the input graph on the left in step (1) yields the graph in the middle, where each edge in the original graph from $i$ to $j$ leads to a subgraph containing from $S_{i,j,k}$ to $E_{i,j,k}$ (the dotted edge denotes an $\varepsilon$-edge). After eliminating all $\varepsilon$-edges in step (2), we obtain a normal graph on the right.

**UnQL** UnQL [3, 4] is a language that was designed for querying unstructured data, and has an expressive power of FO(TC) (first order with transitive closure), with time complexity of answering UnQL query in PTIME. It has a convenient surface language with a select-where structure. Users can write graph queries without explicit use of structural recursion. Since all UnQL queries can be translated into structural recursion which has been shown in [4], it will be sufficient for us to parallelize structural recursion to parallelize the UnQL queries.

### 2.2 MapReduce

The MapReduce programming model [5] was introduced for processing huge datasets on clusters of computers which enables scalable, massively parallel computations. The MapReduce program is executed in a sequence of MapReduce jobs each consisting of a map and a reduce phase separated by an intermediate sort. The map function computes every record of its input data set represented as a key-value pair and outputs a list of intermediate key-value pairs which are then sorted according to their key. The reduce function is invoked for every distinct key together with a list of all according values and outputs a list of values which can be used as input for the next MapReduce job.
3 Graph Query Parallelization

Since we know that all UnQL queries can be translated into structural recursion [4], in this section, we will describe the details of how to parallelize a wide class of structural recursions describing join-free queries by MapReduce.

3.1 Parallelize Structural Recursion

Properties of Structural Recursion The key point that structural recursion can be parallelized is based on the following properties which is refined from the original one in [4].

\[ rec(e)(\{l : g\}) = e @ rec(e)(g) \]

Usually, all UnQL queries except for queries using the join operator can be translated into structural recursion where the expression \( e \) does not depend on the variable \( g \) based on tupling transformation which has been shown in [4]. For all join-free queries, we enjoy the following properties for our parallelization.

\[ rec(e)(g_1 @ g_2) = rec(e)(g_1) @ rec(e)(g_2) \]
\[ rec(e)(\text{cycle}(g)) = \text{cycle}(rec(e)(g)) \]

Now we can show how a query can be evaluated efficiently when translated into structural recursion obeying these rules defined above.

Normally, a graph \( g \) can be expressed as:

\[ g = \&x @ \text{cycle}(g_1 \oplus g_2 \oplus \ldots \oplus g_n) \]

where each sub-graph \( g_i \) can be represented using nine graph constructors including the cycle constructor. This flexible representation makes it possible to deal with the graph database which is too huge enough to be stored in single processing node. The method we described here can also be used for distributed graph database. Now let us see how \( g \) is manipulated by the structural recursion.

\[ rec(e)(g) \]
\[ = rec(e)(\&x) @ rec(e)(\text{cycle}(g_1 \oplus g_2 \oplus \ldots \oplus g_n)) \]
\[ = rec(e)(\&x) @ \text{cycle}(rec(e)(g_1 \oplus g_2 \oplus \ldots \oplus g_n)) \]
\[ = rec(e)(\&x) @ \text{cycle}(rec(e)(g_1) \oplus rec(e)(g_2) \oplus \ldots \oplus rec(e)(g_n)) \]

This suggests us another way of computing \( rec(e) \) on the whole graph by computing each sub-graph first, then combining the results. In fact, we can do this again on each sub-graph. This will surely terminate when the sub-graph is an edge.

Note that we can compute on each edge as a sub-graph, and then combine all the results of each edge. Now considering the popular MapReduce programming model which makes the computing of “map” step followed by “reduce” step, we can exploit the MapReduce model for our graph query processing. In the “map” step, we evaluate structural recursion on each edge by using bulk semantics and eliminate the epsilon edges locally. In the “reduce” step, we eliminate the remain \( \varepsilon \)-edges and remove the unreachable part. We give a detail description of how this will be done.
Parallel Computation with MapReduce Our parallelization can be summarized as the following two steps. The first step is to perform bulk computing on each edge of the whole graph to get sub-graphs and to eliminate $\varepsilon$-edges in each computing node locally. The second step is to eliminate $\varepsilon$-edges and remove the unreachable part by MapReduce iteratively.

\textbf{Bulk Computing and Local $\varepsilon$-edge Elimination}

Recall that structural recursion can be evaluated by bulk semantics which can deal with cyclic graphs. We have implemented the computation of structural recursion in Java. The bulk evaluation function $e$ accepts an edge of the graph $G$ as the input, and results in a set of edges.

We need to explain the types of structural recursion $rec(e(l))$. Let $e$ be a function $e: \text{Label} \rightarrow DB^2_Z$. Label means the input edge label, $DB^2_Z$ denotes the output graph with the same set of input markers and output markers $Z$. For each input edge, it is stored as $(n_i, \text{label}, n_j)$, where $i$ may be equal to $j$ and label cannot be $\varepsilon$-edges, while the result of $e(l)$ is stored as $((\&z_k, n_i), \text{label}, (n_j, \&z_l))$, where $i$ may be equal to $j$, $k$ may be equal to $l$, $z_k, z_l \in Z$, label may be an $\varepsilon$-edge. Here $n_i$ and $n_j$ denote nodes.

Local $\varepsilon$-edge Elimination is based on extended graph bisimulation. We give a formal definition of $\varepsilon$-edge elimination.

\textbf{Definition 1 ($\varepsilon$-edge elimination).} Let $g_1 = (V, E, I, O)$ be an edge labelled graph, with input marker set $X$ and output marker set $Y$, $E \subseteq V \times L \times V$, $L_\varepsilon$ denotes $L \cup \{\varepsilon\}$. After the elimination, we get a labelled graph $g_2 = (V, E', I, O')$. $E'$ and $O'$ are defined as follows:

\[ E' = \{(v, a, v') \mid \exists v_1, v_2, ..., v_n (n \geq 1) \text{ s.t. } v_1 = v, a \neq \varepsilon, (v_1, \varepsilon, v_2) \in E, (v_2, \varepsilon, v_3) \in E, \ldots \} \]
Algorithm 1 preprocessing : bulk computing, local $\varepsilon$-edge elimination

Require: (String key, String edge) {key is not used in the algorithm}
1: set1 = bulkComputing(edge)
2: set2 = localElimination(set1)
3: output all $((v_i, (z_l, v_i), e_{ij}, (v_j, z_m)) \in \text{set2})$

Algorithm 2 iterative − map : edge reversing

Require: (String key, String edge) {key is not used in the algorithm}
1: output $((v_j, \text{reverse } ((v_j, z_m), e_{ij}, (z_l, v_i)))$
2: output $((v_i, ((z_l, v_i), e_{ij}, (v_j, z_m)))$

$O' = \{ (v, \&y) \mid \exists v_1, v_2, ..., v_n (n \geq 1) s.t. v_1 = v, (v_1, \varepsilon, v_2) \in E, (v_2, \varepsilon, v_3) \in E, ...
\}

Graph $g_1$ and $g_2$ in the definition 1 are bisimilar, which shows that our $\varepsilon$-edge elimination is correct. The definition also gives us a way of eliminating the $\varepsilon$-edges by replacing a consecutive sequence of $\varepsilon$-edge followed by a non $\varepsilon$-edge or output marker by the rules in the definition.

We summarize our algorithm in Algorithm 1. Note that we can not eliminate all the $\varepsilon$-edges locally in each computing node. Let us give a typical example for illustration, $e(l)$ compute a sub-graph with all edges are $\varepsilon$-edge. After elimination, there still one $\varepsilon$-edge left because of lack of enough information to decide to eliminate or not. But finally in the second step, we will remove all the $\varepsilon$-edges legally.

Unreachable Node Removal and Global $\varepsilon$-edge Elimination

In order to eliminate $\varepsilon$-edges, we need to compute the transitive closure of $\varepsilon$-edges. The transitive closure computation has been studied intensively[11, 2], but computing transitive closure by MapReduce has not been well recognized[1]. We employed MapReduce for computing transitive closure of $\varepsilon$-edges for acyclic graphs. We also give an approach for cyclic graphs in conclusion.

We remove the unreachable nodes and $\varepsilon$-edges at the same time by using two phases of “map” and “reduce” iteratively. In the “map” phase, we reverse the in-node and out-node of each edge triple, as seen in Algorithm 2. Then in the “reduce” phase, according to the incoming edges and outgoing edges of node $v_j$ to eliminate $\varepsilon$-edges and remove unreachable edges of graph node $v_j$ as described in Algorithm 3.

In “reduce” phase, each reducer computes Algorithm 3. Note that: (1) for each edge pair $((z_l, v_i), e_{ij}, (v_j, z_m)), ((z_m, v_j), e_{jk}, (v_k, z_p))$, if $e_{ij}$ is not $\varepsilon$, and $e_{jk}$ is not $\varepsilon$, output $((z_m, v_j), e_{jk}, (v_k, z_p))$. Otherwise $e_{ij}$ is $\varepsilon$, the answer is based on
Algorithm 3 iterative - reduce : ε-edge elimination, unreachable part removal.

Require: (String key, String edges) {key is v_j, and edges is a set of (v_j, z_m), e_{ijk}, (z_l, v_i)}

1: for each ((v_j, z_m), e_{ijk}, (z_l, v_i)) do
2:   if reversed = true then
3:     output ((z_l, v_i), e_{ijk}, (v_j, z_m)) to set In_j {if reversed, then reverse back}
4:   else
5:     output to set Out_j
6:   end if
7: end for
8: if root(v_j) = true then
9:   process root node
10: else
11:   if size(In_j)=0 or size(Out_j)=0 then
12:     out file Null
13:   else
14:     process node v_j
15:   end if
16: end if

ε_jk. If e_{jk} is ε, then output ( (z_l, v_i), ε, (v_k, z_p) ), else output ( (z_l, v_i), e_{jk}, (v_k, z_p) ) and ( (z_m, v_j), e_{jk}, (v_k, z_p) ); (2) the root node may have no incoming edges, just output ( (z_m, v_j), e_{jk}, (v_k, z_p) ), whose edge label e_{jk} is not ε; (3) nodes except for the root should be checked whether they have incoming edges from the root node. If so, we should filter out the accessible part by markers.

This algorithm terminates when no zero-size set In_j or Out_j is found and no ε-edges exist. The advantage of this algorithm is that when eliminating ε-edges, we delete the unreachable nodes by removing unreachable edges at the same time. Finally, it eliminates all the ε-edges and removes all the unreachable parts. In average case, the complexity is \( O(\log n) \), where \( n \) is the length of the longest path in the graph. It could be \( O(n) \) in worst case, but normally longest path \( n \) of a graph is not big.

4 Implementation and Experiments

We have implemented a parallel query system with Hadoop – an open source MapReduce computing engine. We have confirmed that our parallel graph query algorithm is workable in the real Hadoop cluster with big scale input data.

The Hadoop cluster is constructed with 8 virtual machines (VM) in a cloud computing environment. Each VM has 3 GB memory and one single-core 2.8 GHz CPU. The input data is a randomly generated graph. The labels of edges are all uniquely generated.

Currently, the main purpose of our MapReduce implementation is to show our algorithm is MapReduce-computable for huge graphs, while the performance...
Table 1. Graph Queries on Hadoop

<table>
<thead>
<tr>
<th>Queries</th>
<th>1+3 million</th>
<th>10+30 million</th>
<th>20+60 million</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q₁</td>
<td>104 sec.</td>
<td>545 sec.</td>
<td>1091 sec.</td>
</tr>
<tr>
<td>Q₂</td>
<td>138 sec.</td>
<td>773 sec.</td>
<td>1429 sec.</td>
</tr>
<tr>
<td>Q₃</td>
<td>251 sec.</td>
<td>1073 sec.</td>
<td>1683 sec.</td>
</tr>
</tbody>
</table>

tuning is left as future work. We evaluated the parallel-graph query with three examples:

\[Q₁ = \text{select } \{a : t \} \text{ where } * \cdot a \cdot t \text{ in } db;\]
\[Q₂ = \text{select } \{a : \{ \text{select } b \text{ where } * \cdot b \cdot t₂ \text{ in } \$t₁ \} \} \text{ where } * \cdot a \cdot t₂ \text{ in } \$db;\]
\[Q₃ = \text{select } \{a : \{ \text{select } \{b : \{ \text{select } c \text{ where } c \cdot t₃ \text{ in } \$t₂ \} \} \text{ where } b \cdot t₂ \text{ in } \$t₁ \} \} \text{ where } * \cdot a \cdot t₁ \text{ in } \$db.\]

Q₁ starts from the root of the graph, finds whether there is an edge labelled a, and extracts this edge label and its sub-graph.

Q₂ finds whether two edges labelled a and b respectively are connected in the graph G. Q₃ finds out the linear pattern a.b.c in the graph G.

Now we translate query \(Q₁(db)\) and \(Q₂(db)\) into structural recursion form as \(\text{rec}(\lambda(l).e(l))\).

\[e₁(l) = \text{case } l \text{ of}\]
\[a : (\&z₁ := \{a : &z₂\}, &z₂ := \{a : &z₂\})\]
\[L : (\&z₁ := \{\varepsilon : &z₁\}, &z₂ := \{a : &z₂\})\]

\[e₂(l) = \text{case } l \text{ of}\]
\[a : (\&z₁ := \{a : &z₂\}, &z₂ := \{\varepsilon : &z₂\}, &z₃ := \{\varepsilon : &z₃\})\]
\[b : (\&z₁ := \{\varepsilon : &z₁\}, &z₂ := \{b : &z₃\}, &z₃ := \{\varepsilon : &z₃\})\]
\[L : (\&z₁ := \{\varepsilon : &z₁\}, &z₂ := \{\varepsilon : &z₂\}, &z₃ := \{\varepsilon : &z₃\})\]

L means any other labels except for those listed before L in each \(e₁, e₂\) and \(e₃\) correspond to query \(Q₁\) and \(Q₂\) respectively. We omit the translation of \(Q₃\) since it takes too much space.

The Table 1 shows the execution time. In the head line, \((x+y) \text{ million}\) means the input data are \(x\) million vertices and \(y\) million edges. All three queries can be executed in a fully parallel way and get correct results. The execution time depends on both the size of input data and the longest path in the graph. So the query time is not linear with the data size.

5 Conclusion

We have presented a framework which gives a new way for parallelizing graph queries, by translating UnQL queries into structural recursion and then evaluating structural recursion by MapReduce. Our approach is the first attempt of evaluating structural recursion by the MapReduce programming model. While the experimental results shows its promise, a lot of improvements are still needed. First, we can only process join-free queries for the time now. For a join query, one solution is to firstly evaluate each join-free part using structural recursion and then join the results. Second, for cyclic graphs, the \(\varepsilon\)-edge elimination and
unreachable part removal can be done by treating the cyclic graph $G$ we get after the preprocessing step as an acyclic graph $G'$ plus a set of edges $S$, thus we can translate the computation on cyclic graph to acyclic graph. Third, sometimes the result of a query is just a small part of the whole graph, we can store the graph as a combination of small graphs in a distributive way, reducing data communications among computation nodes.

References