WOOster: A Map-Reduce based Platform for Graph Mining

Aravindan Raghuveer

Yahoo! Bangalore
aravindr@yahoo-inc.com

Abstract

Large scale graphs containing O(billion) of vertices are becoming increasingly common in various applications. With graphs of such proportion, efficient querying infrastructure becomes crucial. In this paper, we propose *WOOster* a hosted querying infrastructure designed specifically for the large graphs. We make two key contributions: a) Design of the WOOster framework. b)Scalable map-reduce algorithms for two popular graph queries: sub-graph match and reachability. Our experiments show that the proposed map-reduce algorithms scale well with large synthetic datasets.

1 Introduction

Graphs are a very intuitive and natural representation for a number of datasets spanning from web structure [2], concepts [3] to molecuar structure [7]. For instance, the purpose of the Web-Of-Concepts (WOC) [3] initiative is to represent the knowledge present in the web as entities and the relationships between entities. A graph is the most intuitive representation of the WOC data with entities being the vertices in the graph and the relationships forming the edges of the graph. We term this graph as the *WOC graph.*Various semantic information sources like automated information extraction systems and feeds from content providers are expected to add and enrich the WOC graph over time.

The WOC graph is intended to reflect the knowledge on the web. Therefore it follows that the WOC graph will have $O(billion)$ vertices and $O(billion)$ edges. In this paper we address the following question: *How to query a graph of this scale?* Faloutsos et.al [5] observe that current graph mining algorithms are not designed to scale to graphs of such magnitude.

Hadoop [1], the open source map-reduce framework is becoming the de-facto standard for analyzing large data sets. Due to its wide acceptance and ease of use, we develop our graph mining algorithms over Hadoop. Current graph mining algorithms assume that random access on the underlying storage system is allowed. With the WOC graph stored on the Hadoop Distributed File System (HDFS) [1], graph mining algorithms based on this assumption will have issues scaling I/O because HDFS is not designed to provide good performance for random access.

As a first step towards a solution, we present *WOOster*, a framework to query the web-of-objects graph. The first prototype of *WOOster* provides the following functionalities:

1. Hosted Solution: In its current form, WOOster is an entirely hosted solution with the user just having to write the graph queries he/she is interested in. The framework translates the query into efficient map-reduce algorithms, executes the query on the WOC graph and notifies the user when the results of the query are ready. In future, WOOster can evolve to also provide a library of map-reduce based graph mining algorithms
2. Backend powered by Map-reduce: We have designed scalable map-reduce algorithms for two important graph queries: Sub-Graph Match and Reachability Query.

- *Sub-Graph Match Query:* This query is used to find all instances of a sub-graph in the WOC graph. The sub-graph match query could be used to mine patterns in the WOC graph.

- *Reachability Query:* This query is used to find vertices reachable from a given vertex and satisfy a set of constraints.

Note that though we develop the system and algorithms for the WOC graph, they are applicable for any graph based mining system. The remainder of the paper is organized as follows. In Section 2, we provide a brief overview about MapReduce. Section 3 discusses the architecture of the WOOster framework and underlying principles behind the WOOster system design. Sections 4 and 5 discuss the proposed map-

17th *International Conference on Management of Data*
COMAD 2011, Bangalore, India, December 19–21, 2011
©*Computer Society of India, 2011*

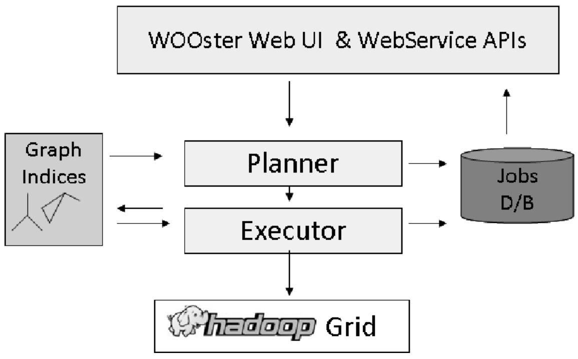


Figure 1: WOOster Architecture

reduce algorithms for Sub-graph search and Reachability query. In Section 6 we discuss the implementation aspects of the WOOster prototype. We study the performance of the Sub-graph search and Reachability query algorithms in Section 7. Section 8 gives a brief overview of the related work done and Section 9 concludes the paper.

2 Background: Map-Reduce

In this section, we briefly describe the google MapReduce [4] software framework. The MapReduce framework is designed to perform distributed computations on large datasets. The programming aspect of the MapReduce framework derives from the map and reduce functions of functional programming. Any program written on the MapReduce framework has two distinct steps: map and reduce. The first step reads input key value pairs and emits key-value pairs. The function that transforms the input key value pairs to the output key value pairs is written by the programmer. Typically, the map step is used to split the problem into parallelizable sub-problems. The framework then groups the values by key and sends it to the next step: reducer. The reducer receives as input a key and the set of values that were associated with that key (as emitted by the map-step). Again, the transformation function of the reducer is written by the programmer. Typically, the reduction step is used to combine the sub-answers (produced by the map-step) to produce the global answer. Hadoop [1] is an open source implementation of the MapReduce framework. The reader is refered to [4] for more details about MapReduce.

3 WOOster: System Overview

We start with two key observations that motivate the proposed design of *WOOster*:

Observation-1: The WOC graph is intended to be the single source of truth about entities and relationships to all the user-facing properties and machine learning algorithms that will use the WOC data for training models. All the users of the WOC graph would need to perform a wide range of querying on the *same underlying dataset*, the WOC graph, to either export a subset of the WOC graph for consumption by their application or to mine patterns.

Observation-2: Graph mining is a computationally and I/O intensive operation, more so given the size of the WOC graph and the number of queries that will access the WOC graph.

Given the above two observations, we take a hosted solution approach to *WOOster* wherein the framework provides query service interface to the users. Please see Figure 1 for the proposed architecture of the WOOster framework. Internally, *WOOster* schedules query jobs to the shared grid, performs batching of queries across multiple users to amortize the computation and I/O cost and maintains indexes on the underlying WOC graph. The wide range of the users for the WOC graph coupled with the cost of executing expensive graph queries individually justifies a dedicated framework like *WOOster* to serve queries on the WOC graph.

Although the hosted solution is the preferred approach, it does preclude the option of *WOOster* providing a library of map-reduce implementations of graph queries.

As shown in Figure 1, the services of WOOster are exposed via a web user interface and web-service APIs. The *planner* modules analyses the set of pending jobs submitted to *WOOster* and creates a plan of execution. The *executor* module executes the plan. As part of executing a query, the *executor* can also collect information to update the indexes. The *Jobs D/B* module stores information about the state of various jobs in the *WOOster* system. The users can query the *Jobs D/B* through the *WOOster* UI. In Section 6 we discuss the details of the prototype of Figure 1 we have implemented.

4 The Sub-Graph Match Query

In this section, we discuss the proposed map-reduce algorithms for performing a sub-graph query. The sub-graph matching problem is stated as follows:
Given the WOC graph G denoted as (V_G, E_G) where V_G is the set of nodes in the graph G and $E_G \subseteq V_G X V_G$ is the set of directed or undirected edges. Each

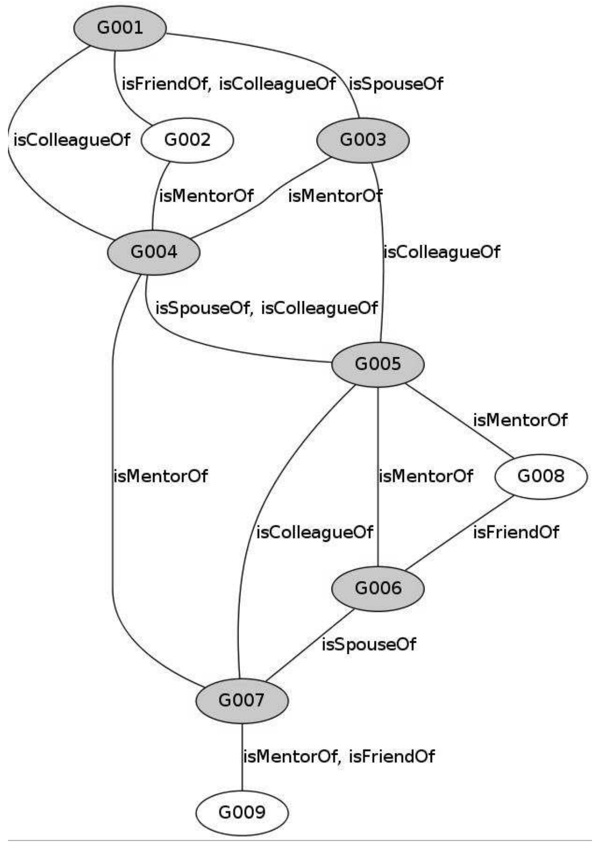


Figure 2: Example Graph: The colored vertices are a match for the query given in Figure 3

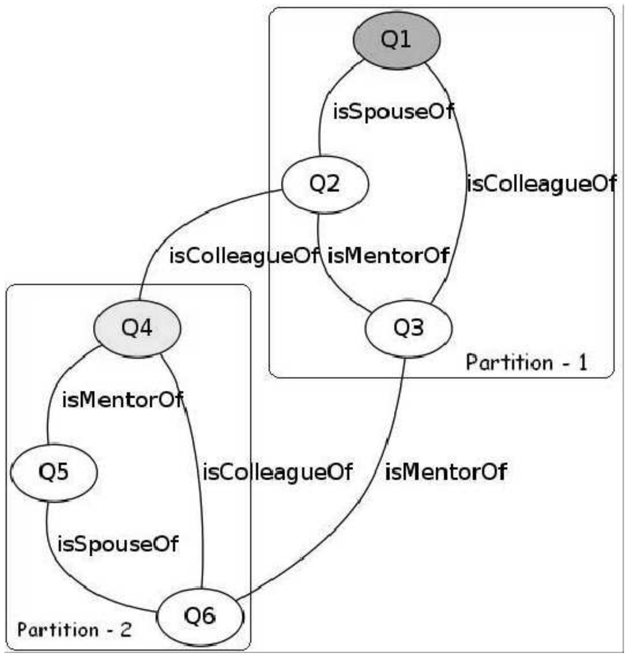


Figure 3: Example Query Graph: Vertices 1,2,3 form a partition for which vertex 1 is the pivot. Vertices 4,5,6 form a partition for which vertex 4 is the pivot.

vertex $v_i \subseteq V_G$ has a set of attributes a_i. Each edge $e_{ij} \subseteq E_G$ between vertex v_i and v_j has a set of edge labels l_{ij}. Figure 2 shows an example graph G as per the definition above. Find all instances of a query graph Q denoted as (G_Q, E_Q), with each vertex $v_i \subseteq G_Q$ having a set of vertex constraints and each edge $e_{ij} \subseteq E_Q$ having a set of edge constraints. Figure 3 shows an example sub-graph query.

Next, we decompose the sub-graph matching problem into two sub-problems in the map-reduce context.

1. How to store the WOC graph on HDFS: What is an efficient way to store the WOC graph on HDFS to enable typical graph queries. See Section 4.1.

2. Scalable Map-Reduce Solution for sub-graph matching: See Section 4.2.

4.1 WOC Graph Storage on HDFS

The WOC graph has two types of information: a) Vertices and the attributes for for each vertex. b) Edges between vertices and the labels for each edge. We assume that vertex attributes are represented as key-value pairs. Each edge in the graph can have one or more labels.

We store with each vertex the adjacency list of the vertex in addition to vertex attributes. This approach may lead to a worst-case O(n^2) storage cost where n is the number of vertices. However we make this trade-off because by providing complete neighbor information at every vertex helps our algorithms described later to make early pruning decisions. Thus each record on the WOC graph dataset will have the following structure: {*vertexGUID, numVertexAttributes, vertexAttributes, numEdges, (neighborGUID:edgeLabels)** }

4.2 Algorithms for Sub-Graph Matching

The proposed solution consists of four steps. The first step (Query Partitioning), splits the query graph Q into one or more query partitions $P_i^Q (i \geq 1)$ and creates a *plan* for the next steps. The second step through the fourth step operate on the WOC graph on HDFS. The second step (Edge Pruning), removes from the search space those edges that do not match any edge in the query graph. The third step (Query Partition Matching), uses the edges found in step-2 to find instances of various query partitions P_i^Q in the graph G. The fourth step (Query Partition Merging), merges the partition instances found in the previous step to form an instance of the query graph as found in the WOC graph. In the next sections, we discuss in detail the algorithms for each step described above.

4.2.1 Notation

We use the following notation in the remaining sections:

1. $\boldsymbol{v_G^a}-$ The a^{th} vertex belonging to the WOC graph G.

2. $\boldsymbol{v_Q^i}-$ The i^{th} vertex belonging to the query graph Q.

3. $\boldsymbol{v_G^a : v_Q^i}-$ A matched vertex which associates the a^{th}graph vertex to the i^{th} query vertex ($v_G^a \leftarrow v_Q^i$).

4. $\boldsymbol{e_x(v1, v2)}-$ An edge connecting the vertices $v1, v2$ where x denotes the graph type to which the edge belongs to. $x \subseteq \{G, Q, M\}$. G represents the WOC Graph, Q represents the query graph and M represents the matched graph to the query (see definition 3 above).

4.2.2 Step-1: Query Graph Partitioning:

By decomposing the query graph into smaller partitions, we are able to parallelize the task of finding matches for each partition. We find the partitions by solving the following optimization problem:

Find the minimum number of partitions of the query graph Q such that the diameter of each partition P_i is at-most 2.

The reasoning for the above formulation is as follows: **Why partitions of diameter 2? :** A partition P with diameter 2 has the following property: The spanning tree of P has a vertex $v_P \subseteq P$ to which all other vertices of P are connected to. We term the vertex v_P as the pivot vertex (or pivot node) of the partition. We leverage on this property of the pivot vertex to design a scalable algorithm to find partition matches. Each graph node sends its information to the pivot node it is connected to. The property that a graph node knows the pivot node is associated with is possible because of the diameter=2 property of the partition. The pivot node collects information from all nodes connecting to it and then decides if the partition is complete in that the partition has all the required nodes and edges. We discussed the proposed partition matching algorithm in more detail in Section 4.2.4.

Why minimize the number of partitions? : The cost of merging the partition instances in Step-4 is proportional to the number of partitions. Therefore we minimize the number of partitions to minimize the merging cost.

Note that the above formulation of the graph partitioning problem is different from the classic graph partitioning problem whose objective is to minimize the edge cut between the partitions. Also this problem does not map to the classic graph coloring problem either.

Solution to the Optimization Problem

We propose the following solution to solve the optimization problem. Let T_i^Q be a minimum spanning tree of the query graph Q. Since the edge weights of the query graph are the same, T_i^Q does not have to be a unique solution. For each T_i^Q, we find the minimum vertex cover $M(T_i^Q)$. Minimum vertex cover$M(G)$ of a graph G is defined as the set of the least number of vertices in G such that every edge $e \subseteq G$ is incident on at least one vertex $v \subseteq M(G)$. We select that $M(T_i^Q)$ which has the least number of vertices and has maximum support across all the spanning trees. The vertices present in the minimum vertex cover are chosen as the pivot vertices.

Figure 3 shows an example query graph. Vertices 1,4 have been selected as pivot vertices for the 2 partitions as described in Figure 3.

4.2.3 Step-2: Early Pruning of non-matching edges

In this second step, we find the set of edges in G that are candidates for edges in Q.

The input to this step is the graph G whose format we defined in Section 4.1. The output of this step is the set of matched edges in graph G that have at least one counterpart in the query graph Q. Also note that when we map a graph edge e_G to a query edge e_Q, the pivot vertex in the graph G for the edge e_G is also automatically fixed. The output of this step is a graph edge in combination with its query edge and the pivot node for edge e_G.

For example, in Figure 2, edges like $(G001, G002)$, $(G007, G009)$ are pruned since they do not have a matching edge in Q. An example of a matched edge is $(G001 : Q1, G003 : Q2)$. Note that a matched edge associates each graph vertex with its counter part in the query.

Algorithms 1 and 2 show the map and reduce algorithms for this step.

Theoretical Scalability Analysis

Note that the bulk of the computation is done in the map step which is parallelized across all the nodes in the cluster. Also note that the output key of the mapper is a combination of an graph edge e_G and its corresponding query edge e_Q. The state space of this key is $\mathrm{O}(N(E_G) \cdot N(E_Q))$, a quantity which scales with the number of instances of Q found in G.

4.2.4 Step-3: Query Partition Matching

Let the set of edges output from the *Edge Pruning* step be $E_{pruned} \subseteq E_G$. The *Edge Pruning* step creates mapping between $E^a_{pruned} \rightarrow E^i_Q$ where $E^a_{pruned} \subseteq E_{pruned}$ and $E^i_Q \subseteq E_Q$. Also to each E^a_{pruned}, the *Edge Pruning* step associates a v^a_P, the pivot node for that edge. The pivot node can be one of the vertices of edge E^a_{pruned}.

The goal of the *Query Partition Matching* step is to form instances of each partition in the graph G. In the example shown in Figure 2, the *Query Partition Matching* step will output the subgraph consisting of

Algorithm 1 Edge Pruning Mapper(v_a^G)

```
1:  Input: $v_a^G :\subset V_G$ .
2:  for all $v_i^Q \subseteq V_Q$ do
3:    if checkVertexMatch($v_a^G, v_i^Q$) then
4:      {Now graph vertex $v_a^G$ can be mapped on to query
        vertex $v_i^Q$. Therefore every query vertex $v_j^Q$ con-
        nected to $v_i^Q$ will have a counterpart graph vertex
        $v_b^G$ connected to $v_a^G$}
5:      Let $v_p^Q \Leftarrow$ the pivot vertex connected to $v_i^Q$
6:      Let $v_p^G$ be the graph vertex that maps to query
        vertex $v_p^Q$ .
7:      for all vertices $v_j^Q$ connected to $v_i^Q$ do
8:        Let $v_b^G$ be the graph node that maps to query
          node $v_j^Q$
9:        Key $\Leftarrow (v_a^G : v_i^Q \Leftrightarrow v_b^G : v_j^Q)$
10:       Value $\Leftarrow v_p^G$
11:       emit Key, Value
12:     end for
13:   end if
14: end for
```

Algorithm 2 Reduce Edge Pruner (key $\leftarrow e_m(v_G^a : v_Q^i, v_G^b : v_Q^j)$, values $\leftarrow< v_G^a : v_Q^p >$)

```
{ /* $e_m$ is a matched edge. Please refer to Section 4.2.1
for definition. }
{ $v_G^a : v_Q^p$ is matched pivot vertex. */}
if size of the values list $geq$ 2 then
  { /* Only in this case do both graph vertices belonging
  to $e_G$ have satisfied vertex and edge constraints */}
  for all $V_p \subseteq$ pivot GUID list do
    Key $\Leftarrow V_p$
    Value $\Leftarrow$ mappedEdge
    Emit (Key,Value)
  end for
end if
```

vertices *G001,G003,G004* as an instance of partition-1 and *G005,G006,G007* as an instance of partition-2.

Now we discuss the map-reduce algorithm for this step and analyze the scalability of the algorithm. The map function of this step emits the pivotVertexID as key and the mappedEdge as value. The reduce step receives as key a pivot graph vertex ID and all the mappedEdges that connect to the pivot Graph vertex ID. The reduce step then emits a partition instance if it finds all the required partition edges in the instance. Partition edges are classified into three types:

Pivot Edge: An edge that connects a non pivot vertex to the pivot vertex (Example in Figure 3: Edge $Q1 - Q2$ in partition-1).

Non-Pivot Edge: An edge that connects two non pivot vertices (Example in Figure 3: $Q2 - Q3$ in partition.)

Cut Edge: An edge that connects a vertex in the partition to a vertex outside the partition (Example in Figure 3: $Q2 - Q4$ in partition).

Algorithm 3 shows the pseudocode for the reduce function.

Algorithm 3 Reduce Query Partition Mapping (key $\leftarrow v_G^a : v_Q^p$, values $\leftarrow< e_m(v1, v2) >$)

```
{/* $v_G^a : v_Q^p$ is matched pivot vertex.
$< e_m(v1, v2) >$ is the set of matched edges that have
$v_G^a : v_Q^p$ as their pivot vertex. */}
partition ID $p \leftarrow$ Get partition ID from pivot vertex $v_Q^p$.
if All pivot edges, non-pivot edges and cut edges of par-
tition $p$ present then
  nodeList $\leftarrow$ set of matched vertices that form an in-
  stance of partition $p$.
  partitonInstanceID $\leftarrow$MD5(nodeList)$\cdot$ p {/* where $\cdot$
  is the string concatenation operator */}
  Let $E_{cut}$ be the set of cut edges in the partition in-
  stance.
  for all $e_G^i \subseteq E_{cut}$ do
    key $\Leftarrow$ (partitionInstanceID)
    value $\Leftarrow$ (nodeList, $e_G^i$)
    emit (key, value)
  end for
end if
```

Theoretical Scalability Analysis

Since the reducer key is the pivotVertexID, the number of reducers will be equal to the number of pivot nodes we identify in the graph G. Each pivot node corresponds to one partition instance. Therefore the parallelism is equal to the number of matching partition instances.

4.2.5 Step-4: Query Partition Merging

The previous step *(Query Partition Mapping)* generates instances of the query partitions P_i found in the graph G. In this step, we iteratively merge adjacent partition instances to eventually form valid instances of the query graph Q_G. While merging a partition instance P_i and P_j, we ensure that instances have matching cut-edges. Only when all the cut edges are present, a merged partition instance is emitted. For example, while merging partitions 1 and 2 in the query graph shown in Figure3, the partition instances of partition 1 and partition 2 should have the same instances of the matched edges $Q2 - Q4$, $Q3 - Q6$.

Theoretical Scalability Analysis

Every iteration of partition merging, consists of two map-reduce programs. Consider an iteration where we are merging partition P_i and P_j. The first map-reduce program (*CutEdgeJoin*) does a join of the cut edges of any partition instance belonging to $\{P_i, P_j\}$. Therefore the parallelism is equal to the number of cut edge instances present in the graph G.

The second program (*VerifyParitionMerge*) compares every partition instance pair created by the previous program and emits a merged partition instances only if all cut edges are present. The degree of parallelism in this program is equal to the number of *(PartitionInstanceID-i PartitionInstanceID-j)* pairs generated by the previous step since it is the map output / reducer input key. The map-reduce signature is omitted for the sake of conciseness. Therefore, similar to the *CutEdgeJoin* program, the degree of parallelism of *VerifyParitionMerge* also scales proportional to Q_G instance matches.

5 The Reachability Query

The reachability query can be stated as follows: Find all instances of the query nodes q_{start} and q_{end} where q_{start} satisfies a set of vertex constraints C_{start} and q_{end} satisfies a set of vertex constraints C_{end} and is no more than D hops away from q_{start}. A special case of the Reachability query is also present in the WOC PRD document where C_{start} is the *GUID* of node q_{start} and C_{end} is not used.

5.1 Proposed Solution

In this section, we propose a map-reduce solution for the reachability query. We first explain the algorithm for a single reachability query and then extend the algorithm for a batched execution of multiple reachability queries submitted by different users.

The proposed algorithm finds all (q_{start}, q_{end}) pairs in D iterations. During every iteration, a path from q_{start} towards q_{end} is extended by one hop. The format of the output of each iteration is the same and is as follows: $\{$*vertexGUID, path-traversed,* C_{end}*, D, queryID, state*$\}$. where:

path-traversed: This shows the hops taken from a q_{start} to the current node. If the number of vertices in this list becomes equal to D, we stop traversing further on this path.

vertexGUID: This is the GUID of the next vertex in the path. The previous field path-traversed, will be extended in the next iteration by one hop to cover vertexGUID.

queryID: This field will be used later for batching multiple reachability queries into one map-reduce task.

state: This field denotes the state of the path-traversed with respect to achieving the target of reaching a q_{end}. If D hops have been reached without reaching a q_{end}, it is set to "NO-MATCH"'. If a match for q_{end} has been found, the state is set to "MATCH". If the search is still in progress, the state is set to "ACTIVE".

Preprocessing: We first find the set of vertices in graph G that satisfy constraint C_{start}. This can be done with a map only task that takes as input the WOC graph G whose format we defined in Section 4.1. The map task reads an input record about a vertex i and applies the constraint C_{start} on the vertex attributes A_i. If the vertex satisfies the constraints C_{start}, then the mapper emits the following record: *key* $\leftarrow$ vertexGUID, *value* $\leftarrow$ (vertexGUID, C_{stop}, D, "placeholder", "ACTIVE"). The penultimate value of the record "placeholder" will be used later for batching multiple reachability queries. For the single query case, all the output records have the same value "placeholder". The last value of the record "ACTIVE" represents the state of the record during the $i = 0^{th}$ iteration. The output dataset of the preprocessing step is denoted as I_0.

The output of the i^{th} iteration is I_i. During the i^{th} iteration the input to the map task is the WOC graph G and the output of the $(i-1)^{th}$ iteration. The mapper emits the GUID of the vertex as key and the remaining values in the record as value. If the input record came from the output of a previous iteration, the record is emitted only if the state is "ACTIVE". The pseudocode of the reducer is shown in Algorithm 4

After D iterations, we can run a filter map job like the preprocessing step to get all the paths that have state=matched. The result set will have all the matching nodes for q_a as the first node of the path-traversed field and q_b as the last node in the path traversed field.

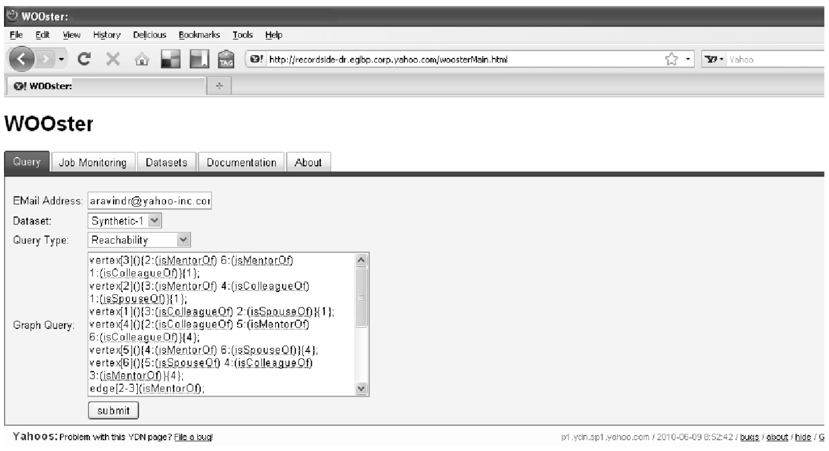
Extension for Batched Execution

1. Instead of the value "placeholder", we emit the Q_i wherever "placeholder" is used.

2. Since the value of D varies from one Q_i to another, we iterate until there is at least one output record that has state="ACTIVE"

6 Wooster: Prototype

In the previous two sections, we described the algorithmic components of the WOOster framework. In this section, we describe the non-algorithmic contributions in the WOOster framework.


WOOster:
File Edit View History Delicious Bookmarks Tools Help
http://recordside-dr.eglbp.corp.yahoo.com/woosterMain.html
Yahoo
WOOster:
WOOster
Query Job Monitoring Datasets Documentation About
submit
Yahoos: Problem with this YDN page? File a bug!
p1.ydn.sp1.yahoo.com / 2010-06-09 8:52:42 / bugs / about / hide / G

EMail Address: aravindr@yahoo-inc.cor
Dataset: Synthetic-1
Query Type: Reachability
Graph Query:

```
vertex[3](){2:(isMentorOf) 6:(isMentorOf)
1:(isColleagueOf)}{1};
vertex[2](){3:(isMentorOf) 4:(isColleagueOf)
1:(isSpouseOf)}{1};
vertex[1](){3:(isColleagueOf) 2:(isSpouseOf)}{1};
vertex[4](){2:(isColleagueOf) 5:(isMentorOf)
6:(isColleagueOf)}{4};
vertex[5](){4:(isMentorOf) 6:(isSpouseOf)}{4};
vertex[6](){5:(isSpouseOf) 4:(isColleagueOf)
3:(isMentorOf)}{4};
edge[2-3](isMentorOf);
```

Figure 4: The WOOster web interface

Algorithm 4 Reducer (key $\leftarrow v_G^a$, values $\leftarrow < records >$)

```
{/* The values list will contain one from the graph $G$. The remaining records from the output of the previous iteration dataset $I_{i-1}$. */}
{/* Let the records from the previous iteration be represented as $I_{i-1}$}
for all record $r \subseteq I_{i-1}$ do
  if vertexGUID satisfies $Q_{end}$ of record $r$ then
    emit record $r$ with the vertexGUID appended to path-traversed state
    continue;
  end if
  if length(path-traversed) of record $r == D - 1$ then
    emit record $r$ with the vertexGUID appended to path-traversed state=“NO-MATCH”.
    continue;
  end if
  {/* This path can be extended one more hop to all the neighbors of the current vertex other than the one from which it came from */}
  for all $v_x$ which is a neighbor of vertexGUID not in path-traversed do
    key $\leftarrow v_x$
    value $\leftarrow$ path-traversed appended with vertexGUID, $C_{stop}$, $D$, “placeholder”, “ACTIVE”
    emit key, value
  end for
end for
```

6.1 WOOster Web Interface

A user can submit a sub-graph match or a reachability query through a web-interface as shown in Figure 4. Once the results of the query are available, the user is notified through email, the location of the results. The results are maintained in a common repository for a week before being purged. On submitting a query, the query is copied to a directory on a hadoop gateway machine.

The *Dataset* tab allows the user to import a new dataset for WOOster to run queries on. The *Job Monitoring* tab allows the user to query the status of his/her job. These two feature are not active yet.

6.2 Planner

The WOOster planner checks a pre-configured directory for new query jobs before launching a job that requires reading of the WOC graph G (example: the mapper step of a reachability query). If new queries have arrived in the directory, it passes the query information to the executor which batches the new query into the current iteration of reading the WOC graph. Through this batching process, we amortize the huge I/O cost of reading the WOC graph.

6.3 Executor

The WOOster executor is responsible for batching all the queries to use the WOC graph. It implements a map task that reads the WOC graph. The executor's map task of the executor acts as a wrapper for the map tasks of the reachability and sub-graph match queries. The executor's map task reads the WOC graph from HDFS and passes it to each sub-graph query that is in step-1 and each reachability query. Through this means, a WOO graph record once read is passed through the various map jobs on the same compute node.

7 Experiments

We study the scalability aspects of the proposed map-reduce algorithm in this section.

7.1 Synthetic Dataset Generation

We use a synthetic graph generator to control the properties of the graph we want to study. The synthetic graph generator creates a graph with a given number of vertices and adds the required number number of edges between randomly chosen pairs of vertices. It also assigns between one to eight attributes to vertices chosen from 100 key value pairs. It assigns between one to eight labels for edges chosen from 100 labels. All our experiments are performed on a graph with 10 million vertices and 50 million edges.

We also generate queries using the following algorithm: We chose a random vertex in the graph and add it to the query. We add a random number of attributes of the vertex as vertex constraints in the query. Next, we randomly chose one of the neighbors of the vertex and add the edge and a randomly chosen edge labels as edge constraints to the query. We hop to chosen neighbor vertex with a probability p_{hop} and repeat the process until we have n nodes in the query graph. The p_{hop} parameter controls the shape of the query graph. $p_{hop} = 0$ gives a star query while $p_{hop} = 1$ yields a linear chain query of n nodes.

7.2 Sub-Graph Match Query

Figure 5 shows the running time of various steps in the sub-graph match query. As we increase the number of reducers, the running time of the Edge Pruning step drops by up to 40%. For the other two steps, we see that the running time remains relatively unchanged. This is because the edge pruning step operates on the entire graph and hence benefits by increasing the number of reducers. The other two steps operate on a subset of the graph and hence do not require as much parallelism as required by edge pruning.

7.3 Reachability Query

In the reachability query experiment, we study the running time for each iteration for a reachability query with $D = 5$. (Refer Figure 6). We make two observations: First, as the number of iterations increase, the running time increases. This is because, as we traverse deeper into the graph, the fanout of the vertices causes more vertices to be processed in the current iteration than the previous iteration. The second observation proves the scalability of the proposed algorithm. As we increase the number of reducers, the running time of the iteration $i = 5$ drops by upto 70%.

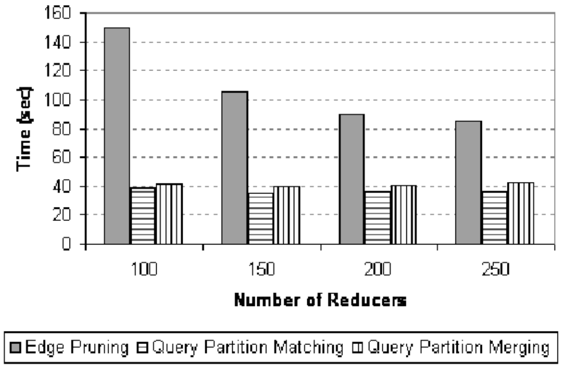


Figure 5: Running Time for the three steps in Sub-graph match. Number of vertices in query = 24, p_{hop}=0.5

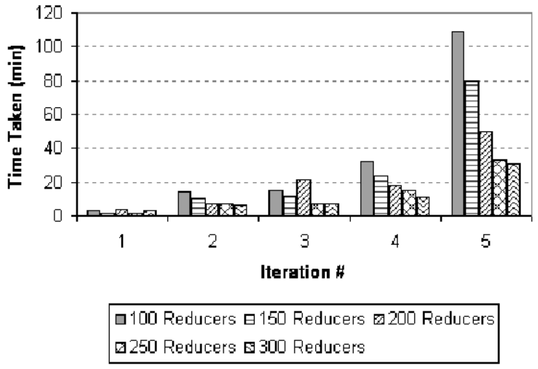


Figure 6: Running Time for each iteration of a $D = 5$ reachability query.

8 Related Work

In this section, we provide a brief overview of related work in graph mining. SubGraph matching and Reachability queries have been studied extensively in literature [10, 8, 9]. A common feature across all the prior work is a novel indexing scheme to represent the graph structure so that queries can be made to run quicker. However most of the prior work do not scale to the proportions we want, at least in their direct form. The Pegasus project at CMU [5] has also started to examine the problem of peta-scale graphs. TALE [8] uses constructs a neighborhood index that is very similar to the data layout that we have chosen for our WOC graph in Section 4.1. However TALE does not directly address the scaling issue that we focus in this paper. Pregel [6] builds on the Bulk Synchronous Parallel model to develop a new framework for large-scale graph processing. Unlike Pregel, Wooster is designed to re-use the Hadoop infrastructure for managing large clusters.

9 Conclusion

*Woo*ster is a framework to efficiently query large scale graphs like the Web of Objects graph. We propose scalable map-reduce algorithms for two important query patterns: sub-graph match and reachability queries. A host of interesting problems remain to be explored in the context of WOOster: what is a good indexing structure that can work well with map-reduce applications? On-board more query types to WOOster. Build a planning module that will also re-use computation within a batch of queries.

References

[1] `http://apache.hadoop.org`.

[2] A. Broder. Graph structure in the Web. *Computer Networks*, 33(1-6):309–320, June 2000.

[3] N. Dalvi, R. Kumar, B. Pang, R. Ramakrishnan, A. Tomkins, P. Bohannon, S. Keerthi, and S. Merugu. A web of concepts. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '09, pages 1–12, New York, NY, USA, 2009. ACM.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[5] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system- implementation and observations. In *In IEEE International Conference on Data Mining (ICDM 2009).*

[6] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing - "abstract". In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 6–6, New York, NY, USA, 2009. ACM.

[7] S. Nijssen and J. Kok. Frequent graph mining and its application to molecular databases. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 5, pages 4571 – 4577 vol.5, oct. 2004.

[8] Y. Tian and J. Patel. Tale: A tool for approximate large graph matching. In *In Proceedings of ICDE 2008.*

[9] S. Trisl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the SIGMOD international conference on Management of data*, pages 845–856, New York, NY, USA, 2007. ACM.

[10] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *ICDE*, pages 966–975. IEEE, 2007.