Horton: Online Query Execution Engine For Large Distributed Graphs

Mohamed Sarwat
Dept. of Computer Science and Engineering
University of Minnesota, Twin Cities
sarwat@cs.umn.edu

Sameh Elnikety Yuxiong He Gabriel Kliot
Microsoft Research
Redmond, WA
{samehe,yuxhe,gkliot}@microsoft.com

***Abstract*—Large graphs are used in many applications, such as social networking. The management of these graphs poses new challenges because such graphs are too large to fit on a single server. Current distributed techniques such as map-reduce and Pregel are not well-suited for processing interactive ad-hoc queries against large graphs. In this paper we demonstrate Horton, a distributed interactive query execution engine for large graphs. Horton defines a query language that allows expressing regular language reachability queries and provides a query execution engine with a query optimizer that allows interactive execution of queries on large distributed graphs in parallel. In the demo, we show the functionality of Horton managing a large graph for a social networking application called Codebook, in which a graph models data on software components, developers, development artifacts (such as bug reports), and their interactions in large software projects.**

I. Introduction

Graphs are widely used in many application domains, including social networking, interactive gaming, online knowledge discovery, computer networks, and the world-wide web. For example, online social networks (OSN) employ large social graphs as used in popular sites such as Facebook [7], and Linkedin [8]. In the simplest form, a *social graph* contains nodes that represent people and edges that represent friendships. Social graphs today are much richer, maintaining data on photos, news, and groups. For instance, a node can represent a person or a photo, and an edge between a person and a photo means that the person is tagged (appears) in the photo.

The popularity and size of social networks pose new challenges. For example, Facebook [7] reports that the number of its users increased five times in a short period, from 100 million in 2008 to 500 million in 2010. Another example is Codebook [1], [2], which is a social network application that maintains information about software engineers, software components, and their interactions in large software projects. Generating the Codebook graph for a large project, such as the Windows or Linux operating systems, results in a very large graph with billions of nodes and edges.

Such a large network cannot be managed on a single server. In addition, current distributed techniques are not well-suited for interactive online querying of large graphs. In particular, the relational model is ill suited for graph query processing [9] making reachability queries hard to express, and therefore distributed database clusters becomes a non-viable option to manage large graphs. The map-reduce [6] framework is designed to process large datasets over a distributed infrastructure, but it is designed for batch processing rather than online query processing. Recently introduced systems for processing large graphs focus on offline batch processing. Systems like Pregel [9] and Surfer [5], [4] support batch processing on graphs with high throughput rather than interactive queries with low latency. We also point out that systems that manage a large number of small graphs, as used in bioinformatics and chemoinformatics, do not meet the requirements for querying large graphs.

With the increased popularity of interactive services such as social network applications, it becomes important to manage large graphs online, supporting querying with small latency. Online processing also allows answering a rich set of ad-hoc queries. In an interactive system, a user may request, for example, to see her friends and their status updates. The user may search for photos in which she is tagged with a specific friend. The database research community has paid little attention to building distributed systems to manage and query large graphs interactively, which is an emerging important application need.

We claim that such challenges require building new systems, specifically designed to handle large graphs. In this paper, we demonstrate Horton: an online distributed query execution engine for large graphs. Horton provides a query language that expresses regular language reachability queries on graphs. Horton consists of a graph query execution engine and a query optimizer that is able to efficiently process online queries on large graphs. As a key design decision for online processing, Horton partitions the graph among several servers, and stores the graph partitions in main memory to offer fast query response time. This is motivated by the availability of many machines in data centers allowing horizontal scaling.

The remainder of the paper is structured as follows. Section II show the graph data model and the query language used in Horton. Section III describes the architecture and design of Horton and the query execution steps, and Section IV presents our demonstration scenario.

II. Horton Data Model and Query Language

In this section we highlight the data model and query language used by Horton.

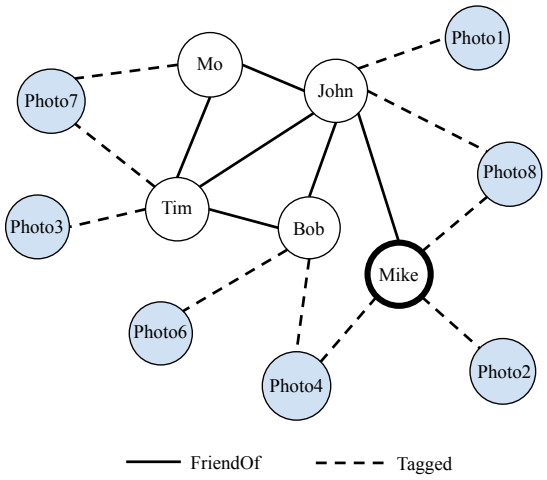

Fig. 1. Example of a small social graph with persons and photos.

A. Data Model

Horton supports rich graph data. Nodes represent entities and have types as well as a set of key value pairs representing the data associated with this entity. Edges represent relationships between entities and have type and data as well. There is no restriction on the kind or amount of data associated with nodes and edges.

Horton manages both directed and undirected graphs. If the graph is directed, each node stores both inbound and outbound edges, to allow queries to traverse both directions. We show examples for undirected graphs as they are simpler, but both are supported in Horton.

B. Query Language

Queries are in the form of *regular language expressions*. A query is a sequence of node predicates and edge predicates. Each predicate can contain conjunctions and disjunctions on node and edge attributes (including type) as well as closures such as regular operators "*" (zero or more), and "+" (one or more).

Figure 1 shows an example of a social graph that has two node types, Person and Photo. In the figure, a solid line edge represents a friendship relationship (i.e., between two Person nodes) and a dotted line edge represents a tagging relationship (i.e., between a Person and a Photo nodes). As a specific query, to retrieve the list of all the photos in which John is tagged, the query is expressed as follows:

```
Photo−Tagged−John
```

To retrieve the photos in which Tim is tagged and at least one of his friends is tagged too, the query is as follows:

```
Tim−Tagged−Photo−Tagged−Person−FriendOf−Tim
```

The query execution engine returns all paths in the graph that satisfy the regular expression. If the user is interested in only a specific part of the path, e.g., a specific node, a SELECT statement is used. For instance, in the last query if the user is interested in photos, the query is expressed as follows:

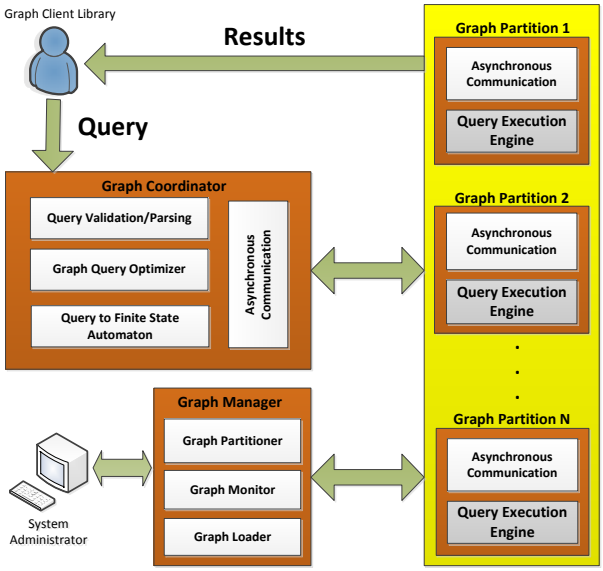

Fig. 2. Horton system architecture.

```
SELECT Photo FROM
Tim−Tagged−Photo−Tagged−Person−FriendOf−Tim
```

III. Horton Architecture

Figure 2 shows an overview of the Horton system. The system comprises four components: the graph client library, graph coordinator, graph partitions, and graph manager. The graph client library sends queries to the graph coordinator and uses an asynchronous messaging system to receive the results. The graph coordinator prepares an execution plan for the query, transforms it into a finite state machine, and initializes the query processing on the appropriate graph partitions. Each partition runs the query executor and sends the query results back to the client. The graph manager transfers the graph between main memory and persistent storage, and partitions the graph.

A. Graph Client library

The graph client library sends queries to the graph coordinator in the form of regular expressions, and receives the query results directly from the graph partitions.

B. Graph Coordinator

The graph coordinator provides the query interface for Horton. The coordinator receives the query from the client library, validates and parses the query. Then, the coordinator optimizes the query plan and translates the query into a finite state machine, which is sent to the partitions for parallel execution. The details are illustrated below:

Query Parsing and Validation. When the graph coordinator receives a query from the graph client, it first parses the query and checks the syntax and the validity of node and edge predicates.

Graph Query Optimizer. The graph query optimizer accepts the query in a parsed regular expression form, enumerates various execution plans, and finds a plan with lowest cost. Cost includes both expected total query execution time on graph partitions and communication cost among them. The Horton graph query optimizer employs a set of optimization strategies and integrates them using a dynamic programming framework to quickly estimate the cost of execution plans. An example optimization strategy is predicate ordering. This optimization strategy evaluates the cost of executing the query with different predicate orders and finds the plan with the lowest cost. Finding a good predicate order to evaluate a query is important because different predicate orders give the same query results but may have orders-of-magnitudes differences in cost due to the sizes of the intermediate results. We use dynamic programming to find the predicate order with the minimum expected cost in time complexity of $O(n^3)$ where n is the number of predicates in the query.

Query to Finite State Machine Translator. After the query is optimized, the query plan is translated into a finite state machine. The finite state machine expresses the query in a form that is efficiently executed by the query execution engine. The state machine is sent to the graph partitions and executed by their local execution engine.

Asynchronous Communication Subsystem. The communication between the graph coordinator and the various graph partitions and among the partitions themselves is done through asynchronous communication protocols that have mechanisms for remote method invocation and for allowing direct streaming of results from a graph partition machine to the client without involving the graph coordinator.

C. Graph Partition

Every graph partition manages a set of graph nodes and edges. Partitions are the main scale-out mechanism for Horton. Each partition resides on a separate server and maintains graph data in main memory. When a graph partition receives the finite state machine of a query from the graph coordinator, it executes the query using a local execution engine. The graph partition may need to communicate with other graph partitions because the execution of a single query may involve distributed processing among several partitions.

Query Execution Engine. Each partition has an execution engine that takes the finite state machine of the query as input and runs a bulk synchronous [10] breadth first graph traversal constrained by the finite state machine. The execution engine checks whether the nodes which are local to the partition satisfy the finite state machine. Next, for all the nodes that satisfy the finite state machine, the execution engine checks if their outgoing edges also satisfy the state machine, and decides whether to continue traversing along the path. When the query execution engine finds a graph node that matches an accepting state in the finite state machine (and therefore satisfies the original query), the execution engine sends this result to the client.

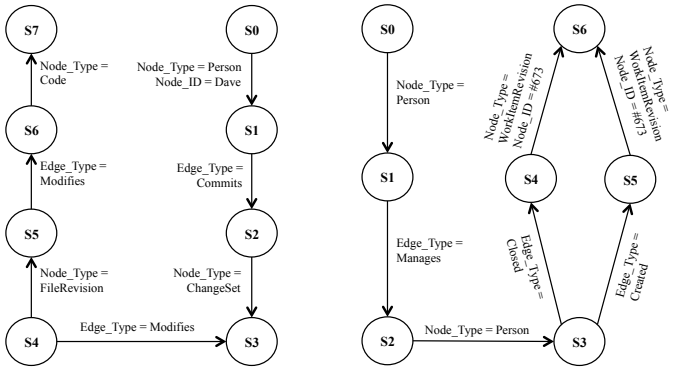

Fig. 3. Finite state machine for queries 1 and 3.

D. Graph Manager

The graph manager provides an administrative interface to manage the graph, to perform tasks such as loading a partitioned graph, and adding or removing servers. Graph manager also supports updates to the graph, such as adding and removing nodes and edges, as well as updating the data associated with nodes and edges.

Horton supports the output of any partitioning algorithm, assigning partitions to servers, and placing nodes in the right partitions. Horton is not equipped with a specific graph partitioning algorithm because this is an expensive offline operation. The simplest form of graph partitioning is hashing based on node or edge attributes. Large graphs are usually scale-free and partitioning algorithms for large scale-free graphs can be used to assign nodes to partitions while preserving locality in graph accesses.

E. Implementation

Horton is written in C# in the .NET framework. Asynchronous communication is implemented using sockets and .NET TPL (task parallel library). Horton is built on top of *Orleans* [3] which is a distributed runtime for cloud applications.

IV. Demonstration Scenario

A. Demonstration Setup

In the demonstration we show the processing of ad-hoc online queries over Codebook graphs, as we choose to use realistic graphs and queries, rather than synthetic data. Codebook [1], [2] is a social network application that represents software engineers, software components, and their interactions in a large software project. In particular, Codebook manages information about source code with its revisions, documentation, and the organizational structure of developers to answer queries such as *"Who wrote this function?"*, *"What libraries depend on this library?"*, and *"Whom should I contact to fix this bug?"*. An example of a Codebook graph is presented in Figure 4.

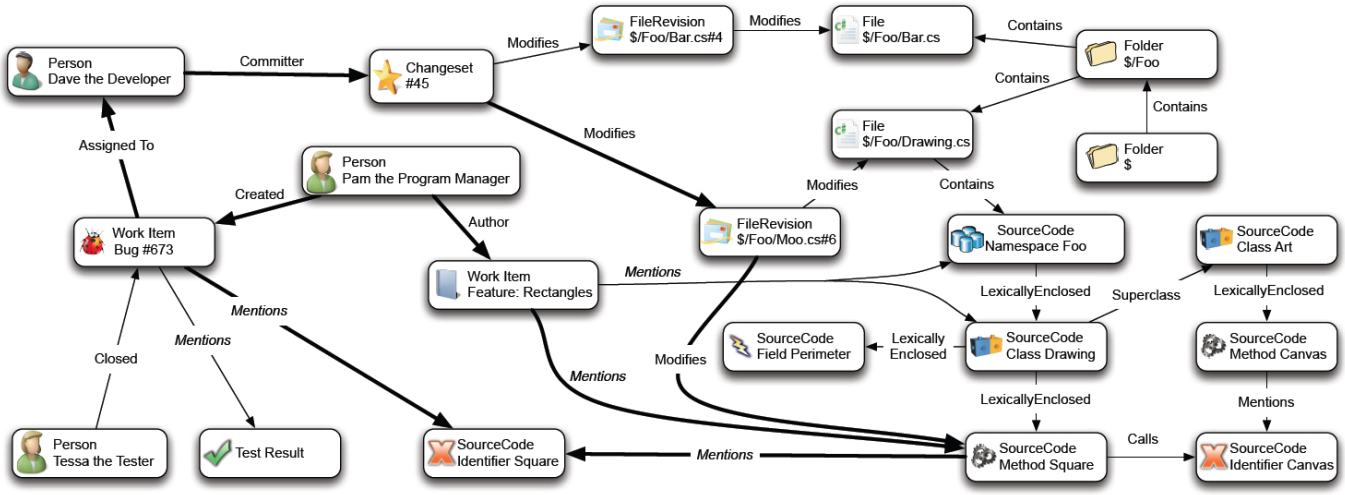

Fig. 4. Example of a Codebook graph [2].

B. Demonstration Scenario

We demonstrate the query processing phases. We issue ad-hoc queries using command line interface. Query examples and their corresponding commands in Horton are given below:

1) Which pieces of source code are modified by *Dave*?

```
C:/> Horton -query "(Person Dave) Committer ChangeSet
Modifies FileRevision Modifies SoureCode"
```

2) Who wrote the specification for the *MethodSquare* code?

```
C:/> Horton -query "(Code MethodSquare) MentionedBy
WordDocument AuthoredBy Person"
```

3) Who is the manager of the person who closed or created work item bug *#673*?

```
C:/> Horton -query "Person Manages Person (Closed |
Created) (WorkItemRevision #673)"
```

The finite state machine for query 1 is shown in Figure 3(a). The start state is $S0$, then the transition from a state to another is conditioned by a node predicate (e.g., *Node_Type = ChangeSet*) or an edge predicate (e.g., *Edge_Type = Committer*). The accepting state is $S7$. Figure 3(b) shows the finite state machine for query 3, which contains a *disjunction* (i.e., Closed OR Created). Optimizations are enabled by flag, *-optimize*, as in the following example:

```
C:/> Horton -optimize -query "(Person Dave) Commits
ChangeSet Modifies FileRevision Modifies Code"
```

The system reports the execution time of each query. The query result is in the form of graph paths (a sequence of graph nodes). For example, the result for query 1 issued on the graph shown in Figure 4 is as follows:

```
Answer Path1:
(Dave) (ChangeSet #45) (FileRevision $Foo/Moo.cs#6)
(SourceCode MethodSquare)
```

We also demonstrate other ad-hoc queries, showing the flexibility of the query language and the effectiveness of the execution engine.

V. Acknowledgement

We thank Alan Geller and Jim Larus for their feedback. We also thank Sergey Bykov, Ravi Pandya, Jorgen Thelin, Andrew Begel, Timothy Cook, and Ron Estrin for their part in building the infrastructure and for many fruitful discussions.

References

[1] Andrew Begel and Robert DeLine. Codebook: Social networking over code. In *Proceedings of the International Conference on Software Engineering, ICSE*, 2009.

[2] Andrew Begel, Khoo Yit Phang, and Thomas Zimmermann. Codebook: Discovering and Exploiting Relationships in Software Repositories. In *Proceedings of the International Conference on Software Engineering, ICSE*, 2010.

[3] Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud Computing for Everyone. In *Proceedings of the ACM Symposium on Cloud Computing, SOCC*, 2011.

[4] Rishan Chen, Xuetian Weng, Bingsheng He, and Mao Yang. Large Graph Processing in the Cloud. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2010.

[5] Rishan Chen, Xuetian Weng, Bingsheng He, Mao Yang, Byron Choi, and Xiaoming Li. On the Efficiency and Programmability of Large Graph Processing in the Cloud. Technical Report MSR-TR-2010-44, Microsoft Research, 2010.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation, OSDI*, 2004.

[7] Facebook. http://www.facebook.com/.

[8] Linkedin. http://www.linkedin.com.

[9] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a System for Large-Scale Graph Processing. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2010.

[10] Leslie G. Valiant. A bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.