A Demonstration of Horton: Online Query Execution on 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,gabriel.kliot}@microsoft.com

ABSTRACT

Graphs are used in many large-scale applications, such as social networking. The management of these graphs poses new challenges as such graphs are too large for a single server to manage efficiently. Current distributed techniques such as map-reduce and Pregel are not well-suited to 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 the expression of 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, whose graph represents data on software components, developers, development artifacts such as bug reports, and their interactions in large software projects.

1. 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), as exemplified by popular sites such as Facebook [1], Twitter [3], and Linkedin [2], employ large social graphs. In the simplest *social graph*, nodes represent persons and edges represent friendships. Social graphs today are much richer, maintaining data on photos, news, and groups. For instance, 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 [1] reports that the number of its users increased from 100 million in 2008 to 500 million in 2010. Another example is Codebook [4, 5], 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 Microsoft Windows operating system, 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 [11], making distributed database clusters a non-viable option to manage large graphs. The map-reduce [9] framework is designed to process large datasets over a distributed infrastructure, but it is explicitly designed for batch processing rather than online query processing. Recently introduced systems for processing large graphs focus on offline batch processing. In particular, systems like Pregel [11] and Surfer [8, 7] support batch processing on graphs with high throughput rather than interactive queries with low latency. We also point out that systems which 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 much richer 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 two specific friends. 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 machines, and stores the partitions in main memory to offer fast query response time. This is motivated by the common availability of many servers in data centers allowing horizontal scaling.

The remainder of the paper is structured as follows. Section 2 describes the architecture and design of Horton and the query execution steps, and Section 3 presents our demonstration scenario.

2. HORTON ARCHITECTURE

An overview of the Horton system is presented in Figure 1. Horton manages both directed on undirected graphs. If the graph is directed, each node stores both inbound and outbound edges. We show examples for undirected graphs and queries only as they simpler, but both are supported in Horton.

The system comprises four logical components: graph client, graph coordinator, graph partitions, and graph manager. The graph client sends queries to the graph coordinator and uses an asyn-

VLDB '11, August 29- September 3, 2011, Seattle, WA

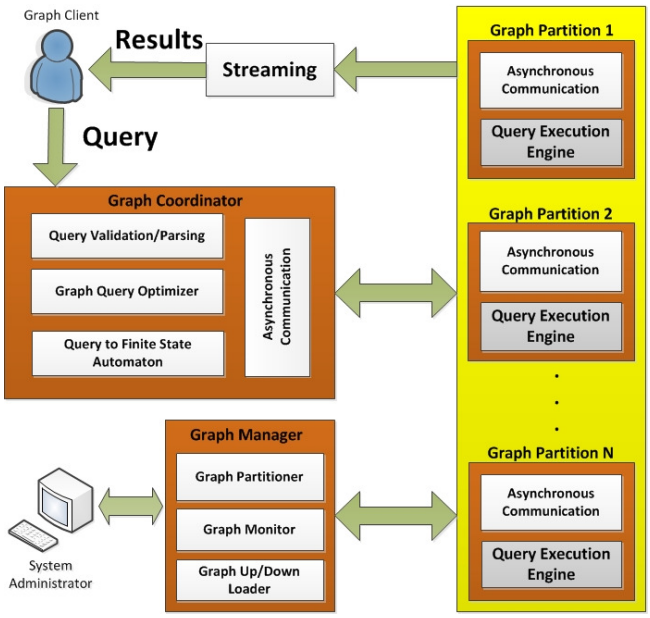


Figure 1: Horton system architecture.

chronous 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 streams the query results back to the client. The graph manager monitors graph activities, transfers the graph between main memory and persistent storage, and partitions the graph.

2.1 Graph Client

The graph client issues the query in the form of a *regular language expression*. The query is a sequence of node predicates and edge predicates, as shown below:

```
(NodePredicate, EdgePredicate, NodePredicate,
...)
```

Each predicate can contain conjunctions and disjunctions on node and edge attributes as well as closures such as regular language operators "*" (zero or more), and "+" (one or more). For example, in the social networking graph shown in Figure 2, if John would like to get the list of all the photos in which he is tagged, the query is expressed as follows:

```
Photo Tagged John
```

If Tim would like to find the photos in which he is tagged and at least one of his friends is tagged too, the query is as follows:

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

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

```
SELECT Photo FROM

Tim Tagged Photo Tagged Person FriendOf Tim
```

After issuing the query, the graph client receives the query results directly from the graph partitions.

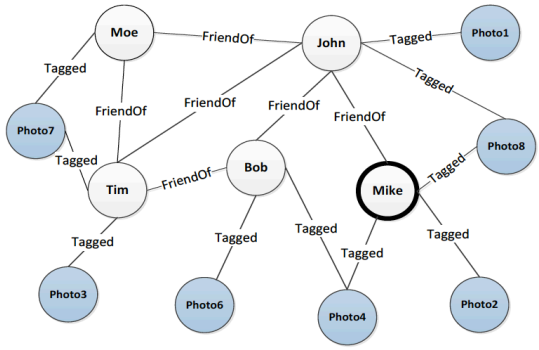


Figure 2: Example of a social graph

2.2 Graph Coordinator

The graph coordinator provides the query interface for Horton. The coordinator receives the query from the client, validates, parses, and optimizes the query. Then, the coordinator translates the query into a finite state machine, and dispatches the query to 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 find a plan with minimal total query execution time and communication cost. *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.

1. Predicates Ordering: This optimization strategy evaluates the cost of executing the query with different predicate orders and finds the plan with minimum cost. Finding a good predicate order to evaluate a query is important because different predicate orders give the same query results but can incur costs that are orders of magnitude higher than the minimal 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.

2. Derived Edges: Derived edges are similar to materialized views in a database system. They store the results of frequently executed queries by adding a derived edge to connect two graph nodes that are indirectly connected to each other. For example, when looking for a friend's friend is a common query, Horton supports creating a class of derived edges representing "friend-of-friend" relationship among people to reduce the cost of executing such queries in the future.

3. Partitioning-aware optimization: In a distributed graph, traversing a remote edge where two nodes of the edge belong to different partitions is more expensive than traversing a local edge where its two nodes belong to the same partition. Horton differentiates them and assigns higher cost to remote edges; the optimization framework uses the weighted cost for remote and local edges in the evaluation of predicate ordering and derived edges.

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

consumed 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.

2.3 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 machine to keep 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 breadth first search [12] constrained by the finite state machine. At each level in the breadth first search, the execution engine performs two steps, namely, a computation step, and communication step. During the computation step, 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 searching 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.

2.4 Graph Manager

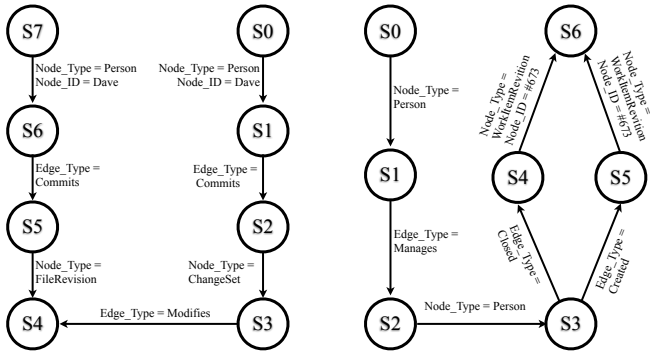
The graph manager provides an administrative interface for the system administrator to manage the graph. The graph administrator can load the graph to the graph partitions. The graph manager provides facilities to add/remove machines, to monitor system performance and overhead metrics.

The simplest form of graph partitioning is hashing. 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. Horton does not compute good partitions because this is an expensive offline operation. It, however, supports the output of any partitioning algorithm, assigning partitions to servers, and placing nodes to the right partitions. We use graph partitioning library that implements the K-way graph partitioning algorithm [10].

2.5 Implementation

Horton is written in C# and uses the .NET framework. Asynchronous communication is implemented using sockets and .NET TPL (task parallel library). Horton is built on top of *Orleans* [6] which is a software framework for building client + cloud applications, and provides a well-designed environment with abstractions that support the development of distributed systems. We deploy Horton using PowerShell scripts on a cluster of machines running Windows Server 2008.

3. DEMONSTRATION SCENARIO



(a) FSM for query 1 (a) FSM for query 3

Figure 4: Finite state machine for queries 1 and 3.

3.1 Demo 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 [4, 5] 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 3.

Since the size of a Codebook graph could be too large, for the demo purposes, we will load only a portion of the graph to a cluster of 40 machines. As a backup plan, in the case of unreliable connection to our cluster, we can load a smaller subset of the graph on few laptops.

3.2 Query Processing

We will demonstrate the query processing phases. The demo attendee can issue an arbitrary query on the graph using our command line interface. We use the command line interface to show that it is easy to write regular language queries and that regular expressions are suitable for graph querying. 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 WordDocumnt 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 demo attendee will see the finite state machine of the query. For example, the finite state machine for query 1 is shown in Figure 4(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*), until reaching the accepting state $S7$. Figure 4(b) shows also the finite state machine for query 3, which contains a *disjunction* (i.e., Closed OR Created).

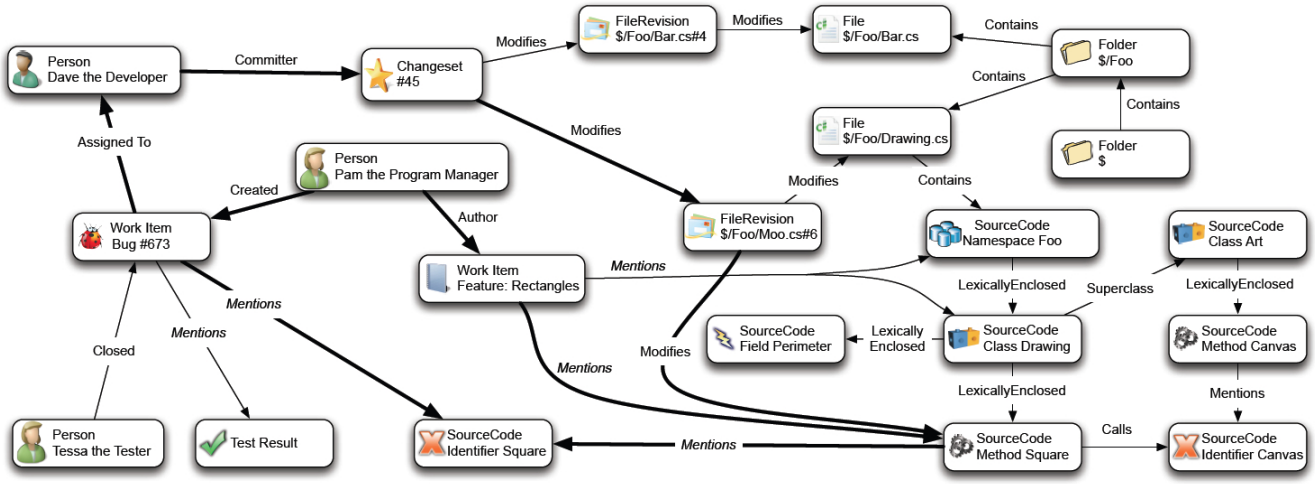


Figure 3: Example of a Codebook graph

We will also show the effects of optimizations by using *–optimize* flag in the query as follows:

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

We will demonstrate the query execution with and without optimizations. With optimizations enabled, we will show the query execution plan resulting from the graph query optimizer. While running the query, the demo attendees will be able to watch the interaction among the graph coordinator and graph partitions residing on different machines. We will also show the execution time of each query, with and without optimizations, to highlight the importance of optimizations. We will then show the query result which the graph client receives. 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 3 is as follows:

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

At the demo, the attendees would be able to issue ad hoc queries to Horton (not restricted to the queries above).

4. 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.

5. CONCLUSION

In this paper, we demonstrate Horton, a query execution engine for large distributed graphs. Horton is designed to interactively query large graphs (i.e., billions of nodes and edges) that are partitioned and managed on many machines. We show how to use Horton with a real social graph (i.e, CodeBook) and interactive queries.

6. REFERENCES

[1] Facebook. `http://www.facebook.com`.

[2] Linkedin. `http://www.linkedin.com`.

[3] Twitter. `http://twitter.com`.

[4] A. Begel and R. DeLine. Codebook: Social networking over code. In *International Conference on Software Engineering (ICSE Companion)*, 2009.

[5] A. Begel, K. Y. Phang, and T. Zimmermann. Codebook: Discovering and Exploiting Relationships in Software Repositories. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010.

[6] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: A framework for cloud computing. Technical Report MSR-TR-2010-159, Microsoft Research, November 2010.

[7] R. Chen, X. Weng, B. He, and M. Yang. Large Graph Processing in the Cloud. In *International Conference on Management of Data (SIGMOD)*, 2010.

[8] R. Chen, X. Weng, B. He, M. Yang, B. Choi, and X. Li. On the Efficiency and Programmability of Large Graph Processing in the Cloud. Technical Report MSR-TR-2010-44, Microsoft Research, 2010.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[10] G. Karypis and V. Kumar. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1996.

[11] 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. In *Proceedings of the international conference on Management of data (SIGMOD)*, 2010.

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