CSC 8711 Finial Project Report

Facebook Graph Search with Neo4j

Team Member:

Jiepeng Zhang

Zhenhua Li

Sha Liu

Instructor: Dr. Raj Sunderraman

Department of Computer Science

Spring 2013

Georgia State University

Atlanta, GA 30302

1 Introduction to Graph Database

A graph database is a storage engine that is specialized in storing and retrieving vast networks of data. It efficiently stores nodes and relationships and allows high performance traversal of those structures. Properties can be added to nodes and relationships.

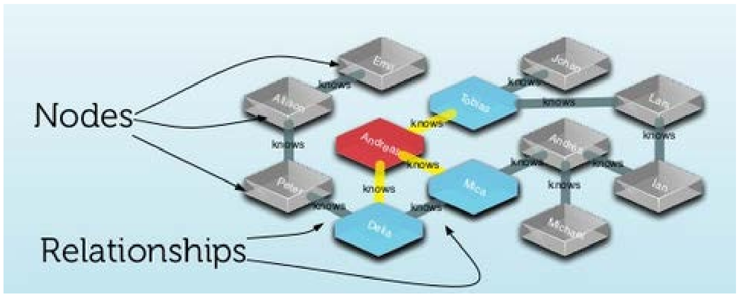


Figure 1 Graph Database

The simplest possible graph is a single node, a record that has named values referred to as properties. A node could start with a single property and grow to a few million properties. Relationships organize nodes into arbitrary structures, allowing a graph to resemble a list, a tree, a map, or a compound entity—any of which can be combined into yet more complex, richly inter-connected structures. To query a graph, we use a traversal, navigating from starting nodes to related nodes according to an algorithm. Also, if we want to find a specific node or relationship according to a property it has, we can use an index to perform a look-up.

Graph database are well suited for storing most kinds of domain models which consist of semi-structured, highly connected data. In almost all domains, there are certain things connected to other things. Semi-structured means that the database does not enforce a schema explicitly, this makes it possible for entities of different types to cohabit. It also means that the domain can evolve and be enriched over time when new requirements are discovered, mostly with no fear of breaking the existing structure.

2. Learning Neo4j

Neo4j is a NOSQL graph database. It consists of nodes and relationships, both of which can have key/value-style properties. Nodes are the graph database name for records, with property keys instead of column names. Relationships are the special part. In Neo4j, relationships are more than a simple foreign-key reference to another record, relationships carry information. So we can link together nodes into semantically rich networks. We could traversal relationships both imperatively using the core API, and declaratively using a query-like Traversal Description. Besides those programmatic traversals there was the powerful graph query language called Cypher and an interesting looking DSL named Gremlin.

Neo4j is a fully transactional database (ACID) that stores data structured as graphs. Proper ACID behavior is the foundation of data reliability. Neo4j enforces that all operations that modify data occur within a transaction, guaranteeing consistent data. Neo4j is also scalable and with high-performance. It can handle graphs with many billions of nodes and relationships on a single server, but can also be scaled out across multiple servers for high availability.

2.1 Nodes, Relationship and Properties

The fundamental units that form a graph are nodes and relationships. In Neo4j, both nodes and relationships can contain properties. Nodes are often used to represent entities. A relationship connects two nodes, has a well-defined, mandatory type and can be optionally directed. Properties are key-value pairs that are attached to both nodes and relationships. When you combine nodes, relationships between them and properties on both nodes and relationships they form a node space – a coherent network representing your business domain data.

Here's how a simple social network might be modeled:

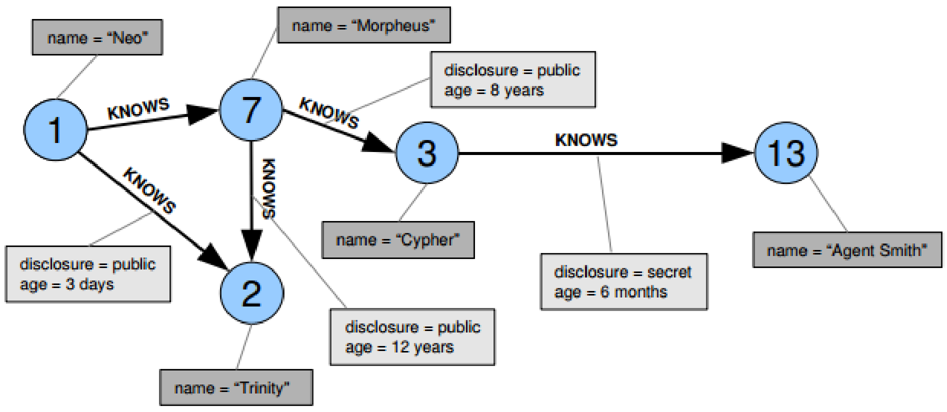
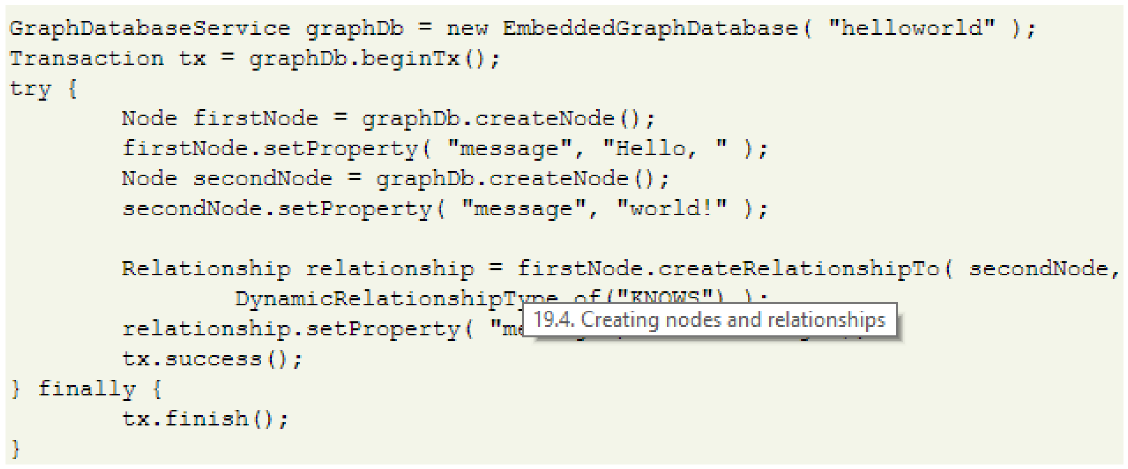


Figure 2 A simple social network graph

You can learn from the model that all nodes have ids and all the relationships have a type. In this case, the only type is KNOWS, a directed relationship representing that one person knows of other. All nodes have a name property and the relationships have properties describing how long the people have known each other and whether the acquaintance is secret.

Let's create two simple nodes and a relationship connects them with codes example.



```
GraphDatabaseService graphDb = new EmbeddedGraphDatabase( "helloworld" );
Transaction tx = graphDb.beginTx();
try {
        Node firstNode = graphDb.createNode();
        firstNode.setProperty( "message", "Hello, " );
        Node secondNode = graphDb.createNode();
        secondNode.setProperty( "message", "world!" );

        Relationship relationship = firstNode.createRelationshipTo( secondNode,
                DynamicRelationshipType.of("KNOWS") );
        relationship.setProperty( "me
        tx.success();
} finally {
        tx.finish();
}
```

Figure 3 Creating Nodes and Relationships

So what we have can be illustrated like this:

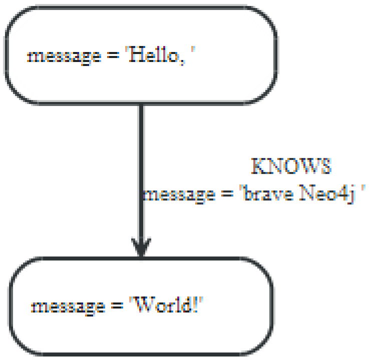


Figure 3 Hello World Graph

2.2 Graph traversal

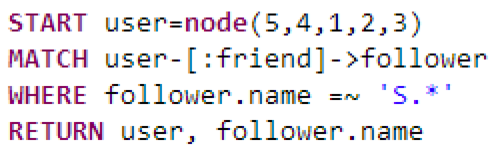
Fast graph traversal of complex, interconnected data and application of graph algorithms are the main use case of a graph database. Traversing a graph means visiting its nodes, following relationships according to some rules. Neo4j comes with a callback based traversal API which lets you specify the traversal rules. At a basic level there's a choice between traversing breadth or depth first. Besides that, we could traversal relationships declaratively using a query-like Traversal Description, such as graph query language Cypher and Gremlin.

2.3 Graph Query Languages

a. Querying the Graph with Cypher

Neo4j provides a graph query language called "Cypher" which draws from many sources. It resembles SQL but with an iconic representation of patterns in the graph.

Cypher queries always begin with a start set of nodes. Those can be either expressed by their IDs or by an index lookup expression. Those start-nodes are then related to other nodes in the match clause. Start and match clauses can introduce new identifiers for nodes and relationships. In the where clause additional filtering of the result set is applied by evaluating expressions. The return clause defines which part of query result will be available. Aggregation also happens in the return clause by using aggregation function on some of the values. Sorting can happen in the order by clause and the skip and limit parts restrict the result set to a certain window.

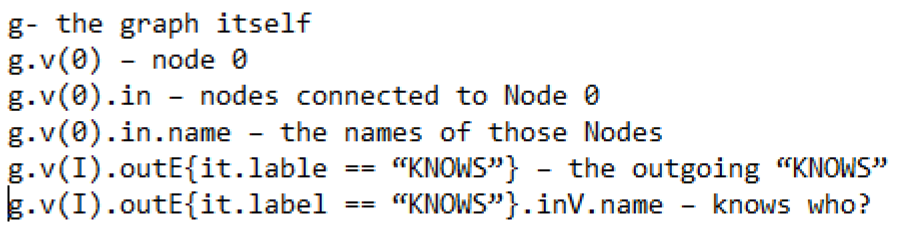


```
START user=node(5,4,1,2,3)
MATCH user-[:friend]->follower
WHERE follower.name =~ 'S.*'
RETURN user, follower.name
```

In this example, we take a list of users (by node ID) and traverse the graph looking for those other users that have an outgoing friend relationship, returning only those followed users who have a name property starting with S.

b. A Graph Traversal DSL Gremlin

Gremlin is an expressive Groovy based Graph Traversal Language. It provides a very expressive way of explicitly scripting traversal through a Neo4j graph. The Neo4j Gremlin Plugin provides an endpoint to send Gremlin scripts to the Neo4j server. The scripts are executed on the server database and the results are returned as Neo4j Node and Relationship representations. Some sample Gremlin queries are as follows:



```
g- the graph itself
g.v(0) – node 0
g.v(0).in – nodes connected to Node 0
g.v(0).in.name – the names of those Nodes
g.v(I).outE{it.lable == "KNOWS"} – the outgoing "KNOWS"
g.v(I).outE{it.label == "KNOWS"}.inV.name – knows who?
```

In this example, g is our graph. v(0) is the node with the id 0. g.v(0).in tells gremlin we want to traverse incoming relationships of node 0. With g.v(0).in.name, we can grab the name property of the nodes we found.

3. **Application Implementation: Facebook Graph Search with Neo4j**

Since the graph database has its advantages in applications of social network search, Facebook Graph Search is a good candidate to explain what it is graph database organized and why it is matters. Here we implemented a Facebook graph search application to prove the concept of how to implement graph search with Neo4j database. Cypher is Neo4j’s graph language and it makes it easy to express what we are looking for in the graph. So Cypher will be used here for the query.

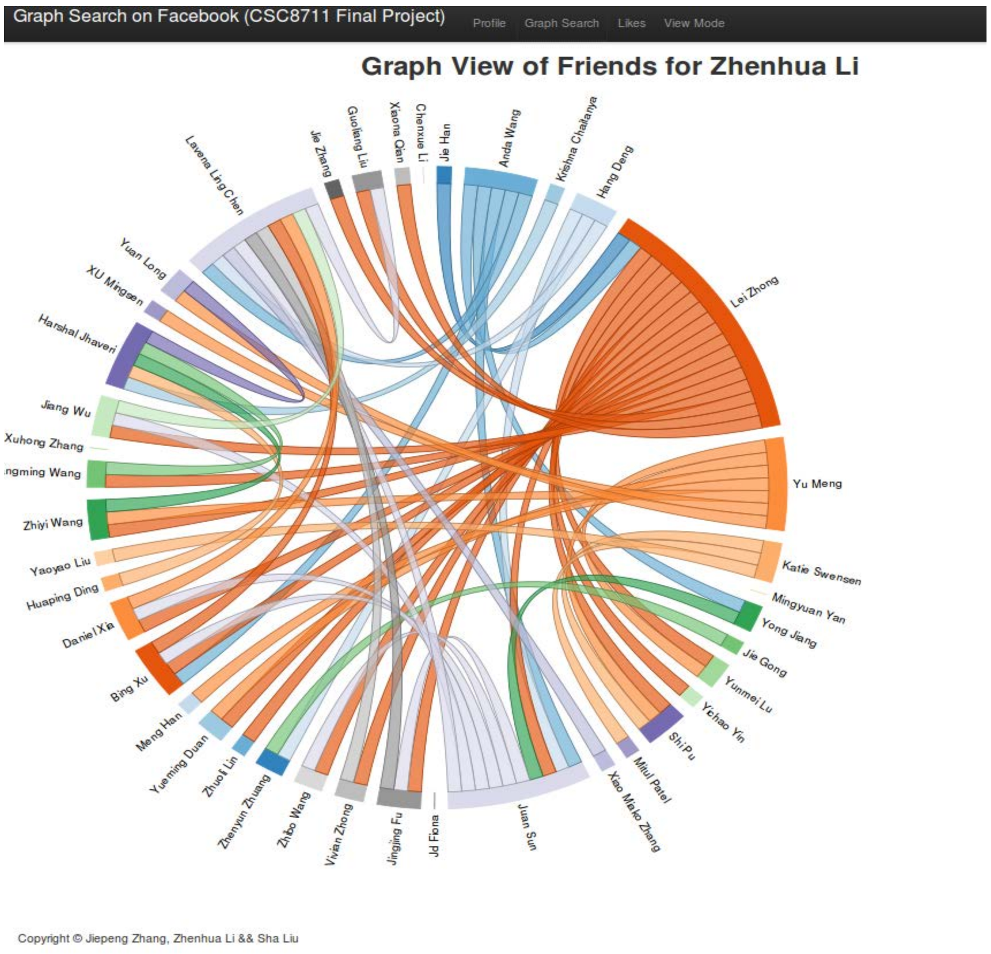
First thing to develop the application, we are creating some rules for things, and the likes relationship, and also the idea of “likes this and that”. The Cypher is run by these rules and a syntax tree is generated with the matching rules. These are then turned into hashes representing pieces of cypher. Looking at the code below you can see how “friends who like Neo4j” gets parsed into Friends, Likes, Thing.

```
class Friends < Treetop::Runtime::SyntaxNode
  def to_cypher
      return {:start  => "me = node({me})",
              :match  => "me -[:friends]-> people",
              :return => "people",
              :params => {"me" => nil }}
  end
end

class Likes < Treetop::Runtime::SyntaxNode
  def to_cypher
      return {:match => "people -[:likes]-> thing"}
  end
end

class Thing < Treetop::Runtime::SyntaxNode
  def to_cypher
      return {:start  => "thing = node:things({thing})",
              :params => {"thing" => "name: " + self.text_value } }
  end
end
```

Then these hashes are combined and turned into a proper Cypher string:

```
class Expression < Treetop::Runtime::SyntaxNode
  def to_cypher
    cypher_hash =  self.elements[0].to_cypher
    cypher_string = ""
    cypher_string << "START "   + cypher_hash[:start].uniq.join(", ")
    cypher_string << " MATCH "  + cypher_hash[:match].uniq.join(", ")
unless cypher_hash[:match].empty?
    cypher_string << " RETURN DISTINCT " +
cypher_hash[:return].uniq.join(", ")
    params = cypher_hash[:params].empty? ? {} :
cypher_hash[:params].uniq.inject {|a,h| a.merge(h)}
    return [cypher_string, params].compact
  end
end
```

Finally we built a Sinatra web application that imports your data from Facebook and a search page as showing in Figure 1.



4. Conclusion

In this project, we introduce neo4j database and also build a small application to show the graph database advantages in applications of social network search. Neo4j is a robust (fully ACID) transactional property graph database. Due to its graph data model, Neo4j is highly agile and blazing fast. For connected data operations, Neo4j runs a thousand times faster than relational databases.

References:

[1] Neo4j Manual

[2] Michael Hunger, “Good Relationships -- The Spring Data Neo4j Guide Book”.