Datalog: A Logical Solution to the SIGMOD 2014 Programming Contest?

Emma Strubell Patrick Verga

Abstract

For our term project we implemented two of the four SIGMOD 2014 programming contest queries each using two different Datalog query evaluation engines, comparing their query execution speed and memory footprint. The two Datalog engines that we compared were Iris [1], the most prominent open-source Datalog engine, and SociaLite [**?**], a new Datalog system with extensions to support queries over web-scale social network graph data which is still in early beta and not yet open-source. We found that, although Iris performed better on the smaller dataset consisting of 1000 users, SociaLite vastly outperformed Iris on the larger dataset of 10,000 users, suggesting that although there is some startup overhead for small amounts of data in SociaLite's implementation, it scales much better than Iris for larger amounts of data. [MEMORY RESULTS]

1 Introduction

For our class project we implemented a solution to two of the four queries making up the SIGMOD 2014 Programming Contest[2] using two different Datalog engines. The aim of the contest was to perform some set of queries as quickly as possible. To do this, we had to create an efficient representation of the data as well as a mechanism for querying that data. The data for this years contest described the complex graph of interactions and entities making up a social network of 10,000+ individuals. Such social networks have become prevalent and bountiful sources of information that organizations can leverage to make useful predictions in many domains, given the computational tools for analyzing these graphs. Methods for efficiently synthesizing the information contained in these representations has become an important focus of research across the study of computer science, including that of database management systems.

As the rules of the contest dictated, our goal was to implement a system able to execute the contest queries as quickly as possible. Due to its lack of expressivity, SQL is not well suited for storing and querying recursive data structures such as graphs. The logical query language Datalog, which natively supports recursion, seems like a more natural solution to the problem of large-scale queries over graphs. However, previous research has found that existing "vanilla" Datalog implementations are still significantly slower than e.g. Java programs performing queries over graphically structured data due to a lack of flexibility provided to the user that is available in traditional programming languages [**?**]. For example, existing Datalog engines insist that graphs be stored as relations despite the fact that adjacency lists are a significantly more efficient representation, and they do not provide mechanisms for the user to specify an evaluation order when a more efficient ordering might be known. Still, the advantages of a high-level, declarative database system such as Datalog over a general purpose programming language like Java are clear, one prominent benefit being automatic parallelization and optimization, but also of course ease of implementation.

To address these shortcomings in existing Datalog implementations, Seo et al. [**?**] describe a set of extensions to Datalog to facilitate efficient social network analysis called SociaLite[3], achieving performance within a few percentage points of highly optimized Java implementations of a number of graph algorithms, and greatly outperforming even commercial Datalog systems. They also have

[1] `http://www.iris-reasoner.org/`

[2] `http://www.cs.albany.edu/~sigmod14contest/task.html`

[3] `http://socialite-lang.github.io/`

followed up on that original publication with additional extensions for using the MapReduce model in order to scale to even larger quantities of data [**?**]. Unfortunately, the authors have not yet made their code open source (though they cited a mid-April date), but they did provide us with a compiled binary upon request. We implemented contest queries 2 and 3 using SociaLite as well as the open-source Datalog engine Iris[4] to evaluate (1) Datalog's proficiency in general at computing these types of queries and (2) the increase in query evaluation speed, if any, resulting from SociaLite's extensions to Datalog on the contest data. We found that, although Iris performed better on the smaller dataset consisting of 1000 users, SociaLite vastly outperformed Iris on the larger dataset of 10,000 users, suggesting that although there is some startup overhead for small amounts of data in SociaLite's implementation, it scales much better than Iris for larger amounts of data. We also compared memory footprints between the two implementations, finding Socialite to be much more memory efficient.

2 Problem description

This years SIGMOD contest posed four realistically complex queries over the given social network data. The contest provided two datasets simulating that of a large social network, generated by LDBC's social network benchmark generator[5], one "small" dataset made up of a network of 1000 individuals, and a "large" dataset of 10,000 users. The contest also evaluated entries on a "huge" dataset, but that data has not yet been made available.

– insert description of schema here

Due to time constraints and the overhead of performing a comparison between systems on top of implementing the queries themselves, we only implemented two of the four queries: Query 2 and Query 3. We also selected these queries because they were the two that could be performed the most within the constraints of the Iris system, which lacks recursive aggregate queries (required to compute shortest paths within a query).

The two queries we did not implement were Query 1 and Query 4. Query 1 asks for frequent communication paths of shortest distance in the graph. To perform this query, we need to find the shortest path between the two given individuals subject to some constraints over their communication metadata. Query 4, is to find the most central individuals in the network. In other words, we must find the people who are the most central or connected nodes (using the metric of betweenness centrality) in a subgraph derived by some constraints such as membership in a specific forum. As in Query 1, computing betweenness centrality also requires computing the shortest path between each pair of individuals in the graph. The second and third queries, which we implemented for this project, are described in more detail below.

2.1 Query 2: Interests with Large Communities

The goal of the second query is given some integer k and date d, to find the k interests with the largest underlying communities of users born on or after d.

To perform this query we required four tables:

- person(id, birthday)
- tag(tag.id, tagName)
- person_hasInterest_tag(person.id, tag.id)
- person_knows_person(person1.id, person2.id)

2.2 Query 3: Socialization Suggestion

The third query is to find closely connected pairs of people who are in the same location and share some number of interests, the goal being to recommend connections based on mutual interests. Given an integer k, integer h, and String p, find the k pairs of people who are associated with place p and are within h degrees of seperation.

To perform this query we required ten tables:

- place(id, name)
- place_isPartOf_place(place1.id, place2.id)
- person(id)
- person_hasInterest_tag(person.id, tag.id)
- person_isLocatedIn_place(person.id, place.id)
- person_workAt_organisation(person.id, organisation.id)

[4] http://www.iris-reasoner.org/

[5] https://github.com/ldbc/ldbc_socialnet_bm

- person_studyAt_organisation(person.id, organisation.id)
- person_knows_person(person1.id, person2.id)
- organisation(id)
- organisation_isLocatedIn_place(organisation.id, place.id)

3 Datalog evaluation engines

3.1 Iris

Iris (Integrated Rule Inference System) appears to be the most prominent (and well-documented) open source Datalog query evaluation engine, cited in [**?**], so we selected it as a baseline system to which to compare. – talk about how it doesn't have aggregates (neither recursive nor non-recursive) so we had to implement that stuff in Java, theoretically giving Iris an advantage – talk about reading in data

3.2 SociaLite

The three main extensions with which the authors achieve these gains are (1) tail-nested tables; (2) recursive aggregate functions; and (3) user-guided execution order. Tail nested tables address the issue of data representation, and allow for graph data to be represented efficiently as what are essentially adjacency lists rather than as traditional relations. Native support for certain recursive aggregate functions allows for common graph queries, such as those based on path length, to be expressed succinctly and to be evaluated efficiently using semi-naive evaluation. Finally, providing a mechanism for users to specify an evaluation order over vertices can significantly decrease the runtime required to traverse a graph if the graph is known to have a certain structure.

– talk about actual system, i.e. Python/Jython

4 Methods

We implemented and evaluated Query 2 and Query 3 each on both the Iris and SociaLite query evalutation engines. All evaluation occurred in memory in both systems due to the limitations of the software packages, and rules were evaluated using semi-naive evaluation, which is the only strategy supported by Iris (in order to facilitate recursive aggregate functions). Both engines are implemented in Java, athough we were able to interact closely with the Java API in the case of Iris, we could only use SociaLite via its Python interface. In both cases we used Java to dynamically generate the queries given the inputs to the query (e.g. maximum number of hops) in each package's dialect of Datalog, which we then provided to the engine to evaluate. Any required post-processing was then performed in Java.

– describe actual queries here – give example of each

Query 3 asks for the top k pairs of individuals connected by at most h hops who live, work or go to school in the same location, ranked by number of shared interests. We solved this problem by writing a function that generates a Datalog query given the inputs k, h and location. The following generated Datalog query provides an intermediate solution, lacking aggregation, to the contest query: query3(1, 2, Amherst):

```
all_locs(locid) :- place(locid, Amherst, _, _)
all_locs(locid) :- all_locs(parentlocid),
      place_isPartOf_place(locid, parentlocid),
          place(locid, name, _, _)

all_orgs(orgid) :- organisation(orgid, _, _, _),
          organisation_isLocatedIn_place(orgid,
               locid),
      all_locs(locid)

loc_people(pid) :- person_isLocatedIn_place(pid,
     locid), all_locs(locid)

org_people(pid) :- person_workAt_organisation(pid,
     orgid, _), all_orgs(orgid)
org_people(pid) :- person_studyAt_organisation(pid,
      orgid, _), all_orgs(orgid)

all_people(pid) :- loc_people(pid)
all_people(pid) :- org_people(pid)

hop1(pid0,pid1) :- person(pid0, _, _, _, _, _, _, _)
     ,
      person_knows_person(pid0, pid1),
      person(pid1, _, _, _, _, _, _, _)
hop2(pid0,pid2) :- person(pid0, _, _, _, _, _, _, _)
     ,
      person_knows_person(pid0, pid1),
      person(pid1, _, _, _, _, _, _, _),
      person_knows_person(pid1, pid2),
      person(pid2, _, _, _, _, _, _, _)

all_hops(pid1,pid2) :- hop1(pid1,pid2)
all_hops(pid1,pid2) :- hop2(pid1,pid2)
```

```
common_interests(pid1,pid2,'') :- all_hops(pid1,
    pid2),
   all_people(pid1),
   all_people(pid2)
common_interests(pid1,pid2,interest) :-
    common_interests(pid1, pid2, ''),
        person_hasInterest_tag(pid1,interest),
        person_hasInterest_tag(pid2,interest)
```

query3{example.py

Query performed: `common_interests(pid1,pid2`

The above produces all triples of people and interests that satisfy the location and hop constraints, including duplicates and self-pairs. We took this intermediary result and aggregated the top k counts of shared interests between non-duplicate pairs of users.

5 Results

Queries were run on a machine with a 3.4 GhZ Intel Core i7 processor and 16GB RAM, which is comparable to that used for evaluating contest entries (8 CPU cores and 16GB RAM). All queries returned the correct results.

Figure 1: This is the caption

Figure 2: This is the caption

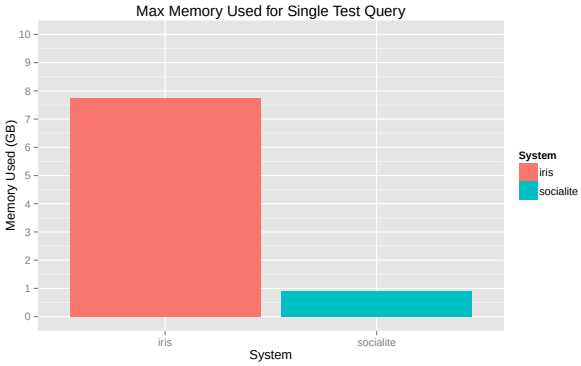

Figure 3: This is the caption

6 Discussion