Final Report

James Atwood and Luis Pineda

May 6, 2014

1 Introduction

Our project was a submission to the SIGMOD 2014 programming challenge, where teams are provided with a large (relational) social network dataset and are asked to implement four queries related to the graph structure of the data. Using a relatively simple design and incremental optimization, we were able to complete our Java implementation by the April 15th deadline and submit it for evaluation. According to the leaderboards[1], which provide preliminary results, our entry ranks 22nd out of 32 total submissions. Unfortunately, the final evaluation on held-out data will not be released until after this report has been written, so we can not report our official standing.

The motivation for choosing this project was twofold; first, there is a large (and quickly increasing) volume of graph-structured data available today, and second, developing efficient mechanisms for representing and querying graph data is a challenging research problem that is currently the subject of considerable interest in the database community.

2 Background and Related Work

2.1 Challenge Description

2.1.1 Data

We are provided with a relational dataset which describes a social network. Example entities include people and places and example relations include 'knows' and 'lives at'. Each entity and relationship is stored as a pipe-delimited file. Entity files are named after the entity type (e.g. 'person.csv') and contain features. Relation files are named after the relation they contain (e.g. 'person_knows_person.csv') and contain pairs of entities which have that relation.

2.1.2 Task

The task is to return the correct results of a provided set of queries against the provided data as quickly as possible. Performance is first measured by

[1] `http://www.cs.albany.edu/~sigmod14contest/leaders.html` under team name 'shparg'

correctness (if any query results are incorrect, a submission is invalid) followed by runtime (with lower being better). ‘Runtime’ means the wall-clock time from program initiation to termination. Note that subtasks like reading the data into memory or constructing an index will factor in to the runtime.

2.1.3 Query types

There are four types of queries that need to be answered.

2.1.4 query1(p1, p2, x)

Given two integer person ids p1 and p2, and another integer x, find the minimum number of hops between p1 and p2 in the graph induced by persons who

1. have made more than x comments in reply to each others’ comments, and
2. know each other.

2.1.5 query2(k, d)

Given an integer k and a birthday d, find the k interest tags with the largest range, where the range of an interest tag is defined as the size of the largest connected component in the graph induced by persons who

1. have that interest,
2. were born on d or later, and
3. know each other.

2.1.6 query3(k, h, p)

Given an integer k, an integer maximum hop count h, and a string place name p, find the top-k similar pairs of persons based on the number of common interest tags. For each of the k pairs mentioned above, the two persons must be located in p or study or work at organisations in p. Furthermore, these two persons must be no more than h hops away from each other in the graph of people who know each other.

2.1.7 query4(k, t)

Given an integer k and a string tag name t, find the k persons who have the highest closeness centrality values in the graph induced by persons who

1. are members of forums that have tag name t, and
2. know each other.

2.2 Related Work

Traditionally, research in databases has focused on the relational model first proposed by Codd [2]. This model becomes awkward and inefficient when applied to graph data [6], particularly for queries related to complex structure (i.e., requiring more than nearest neighbors). For an example, please see [3] Figures 1 and 2. More recent work has proposed other data models and query languages that more appropriately capture the rich structure evident in graph data; for instance, [3, 7, 5] (see [1] for a survey of recent graph database models).

3 Implementation

3.1 Overall Approach

We divided the challange into two subtasks: first, read and index all of the data that is required to answer the queries (the 'reading phase'); then, perform the queries and output results (the 'query phase'). We found that both tasks' runtime was on the same order of magnitude.

3.2 Data Representation

3.2.1 Embedded Database Design

Originally, we designed our implementation around the open-source Neo4j[2] disk-based graph database system. We thought this system was appropriate because the queries in the challenge are largely path-oriented, and it is unlikely that all of the relevant data will fit in memory. Neo4j makes use of the *ADI* index structure [4, Chapter 6] described in [8], which is designed to facilitate efficient edge support checking (that is, quickly finding edges) and adjacent edge checking (that is, quickly finding edges that share a node). This index structure seemed well-suited to the task at hand because it allows a graph on disk to be efficiently queried with regards to path.

Our implementation of this approach performed very poorly, however. An implicit assumption of this design was that the high fixed cost of populating the graph database would be amortized over the large number of queries against the data. Instead, we found that the runtime cost of reading and indexing the data using Neo4j was prohibitively large, therefore any potential improvement Neo4j could offer in query performance would not offset the cost of populating the database. It seems likely that this undesirable behavior will be found with other database systems; if fixed setup costs are amortized over the lifetime of a database system measured in years, the cost of establishing the database is trivial, so reducing this cost is likely not a design goal.

[2]`http://www.neo4j.org/`

3.2.2 In-Memory Design

We turned our attention to a simpler key-value approach tailored to the SIGMOD challenge dataset. Specifically, we indexed nodes and edges via several simple in-memory hash tables, one for each type of node or edge relevant to the queries. This design choice was motivated by an analysis of the provided datasets (1k and 10k persons), which suggested that most of the storage needs are due to node types whose persistence is not needed to answer the queries (i.e., comments and forums). With the right pre-processing this information is only needed to update information only while the database is populated (e.g., how many comments have persons given to each other). We suspect that, at least for the 100k persons dataset, the relevant part of the graph should fit in the 15Gb of available memory for the contest (our current implementation uses approx. 800Mb for the 10k dataset). Overall, the current memory-based approach both reduced the time it took to index the data and improved the speed of the queries by orders of magnitude with respect to our initial Neo4j implementation.

This memory-based design, implemented in Java, defines data structures for each of the relevant entities defined in this database: person, tags, places and forums. We also created a specialized edge structure for the person_knows_person relationship that, besides the pointer to the person nodes, also stores the number of comments that each of the persons have replied to the other (this is very useful to answer query 1 efficiently).

Using these structures, our current pipeline is as follows. When the program starts, a hash table for each edge and node type is populated. In particular, the following hash tables are maintained:

- A person node table indexed by a 64-bit id number.
- A tag node table indexed by a 64-bit id number.
- A place node table indexed by a 64-bit id number.
- A forum node table indexed by a 64-bit id number.
- A commentIDCreatedBypersonID table indexed by a 64-bit id number.
- A organizationIDLocatedAtPlaceID table indexed by a 64-bit id number.
- A placeIDLocatedAtPlaceID table indexed by a 64-bit id number.
- A placesIDWithNames table indexed by a string number specifying the place name. The value is a list of placeIDs that have that name (currently represented as a space separated string of place IDs).

These tables are populated by reading files such as person.csv, comments.csv, tags.csv, comment_hasCreator_person.csv and so forth. Each node type stores edges to other nodes whenever there is a relationship that is useful for answering at least one of the queries. Specifically, the following edges are stored:

- A person node stores: a list of edges, one for each known person, a list of pointers to tag nodes the person is interested in, and a list of pointers to place nodes the person is located at.
- A tag node stores: a list of pointers to interested person nodes and a list of pointers to persons nodes that are members of forums with this tag.
- A forum node stores: a list of pointers to tag nodes with the interests of the forum.

3.3 Query Implementations

We implement each query as a graph algorithm over the graph defined by the indexed data.

3.3.1 Query1

A breadth first search (BFS) of the person_knows_person graph based on the constraint on number of replies x. Edges with less than x replies are pruned by the BFS.

3.3.2 Query2

Add all tags to a priority queue where the order of the tags is based on the size of the largest connected component of the induced graph. This uses the list of persons that are interested in the tag, and information about the birthday stored by each person node. With this information, the induced graph is created on-the-fly and the size of the connected component is computed using a BFS.

3.3.3 Query3

placesIDWithNames is used to find all placeIDs with the given name. For each of these places p, we do a linear scan to find all persons located at p and add these persons to the induced graph. To check if a person is located at p, we use both the list of places stored by the person node, but also the table placeID-LocatedAtPlaceID to recursively check if any place in the induced hierarchy is contained in p. If at least one does, the person is added to the induced graph.

When all the relevant persons are added to the graph, the similarity score of all possible pairs of persons in this graph is computed and these are added to a priority queue. To speed up the similarity computation, the list of persons interested in that tag node is indexed by person id. Therefore the similarity between two persons can be computed in linear time on the number of interest tags.

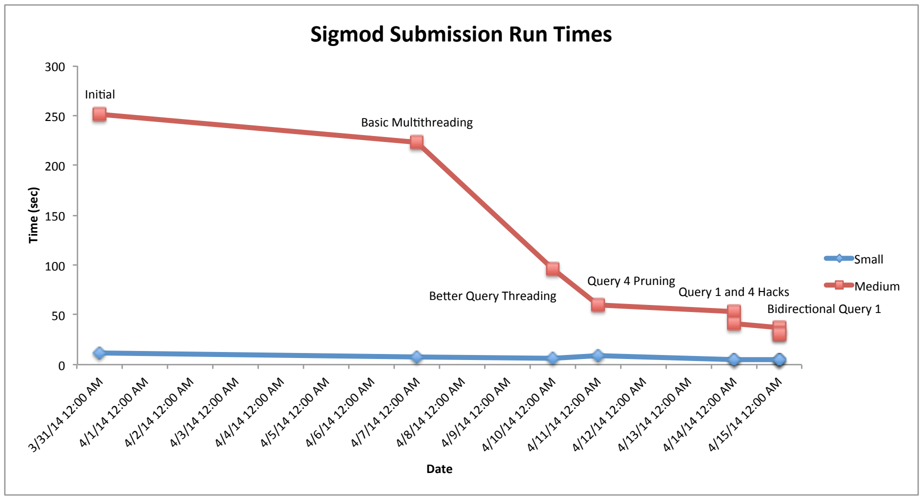

Figure 1: Evolution of the performance of our implementation on the SIGMOD system. The horizontal axis gives the date of the submission and while the vertical axis shows the runtime as reported by the submission system. Each submission is annotated with the optimizations that were introduced with it. 'Medium' indicates the dataset with 10,000 people and 'Small' the dataset with 1,000 people.

3.3.4 Query4

First a linear search is used to find the tag with the given name. Then the induced graph of persons that are member of forums with this tag is created, using the list stored by each tag node.

On the induced graph the centrality score of each person is computed by using a BFS. Every time a node is expanding during the BFS, the algorithm checks if the best possible centrality this person can achieve is smaller than the k-th best centrality seen so far. If it is, the BFS is stopped.

3.4 Concurrency

We introduced concurrent behavior in order to take full advantage of the multi-core test system (which is described in section 4). All concurrent behavior took place in the query phase. Specifically, the queries were distributed evenly among available cores and run concurrently, and within queries of type four, the centrality of each individual was computed concurrently.

4 Results

Our implementation's performance is shown in Figure 4. All times were provided by the SIGMOD submission system. According to the challenge description[3], performance was measured on a server with the following specification:

- Processors: Two 2.67 GHz Intel Xeon E5430 (4 cores each, 8 cores total)
- Main Memory: 15 GB
- OS: Red Hat Enterprise Linux Server 6.5 (Santiago)
- Java: JDK 1.7.0

References

[1] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[2] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[3] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418. ACM, 2008.

[4] J. W. Ian Robinson and E. Eifrem. GraphDatabases. pages 1–223, May 2013.

[5] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[6] M. A. Rodriguez and P. Neubauer. The graph traversal pattern., 2011.

[7] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

[8] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable mining of large disk-based graph databases. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–325. ACM, 2004.

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