Final Report

James Atwood and Luis Pineda

May 8, 2014

1 Introduction

Our project was a submission[1] to the SIGMOD 2014 programming challenge. In this competition 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[2], which provide preliminary results, our entry ranks 22nd out of 32 total submissions (see Figure 1). 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, interest tags and places, and example relations include 'person knows person', 'person works at place' and 'person has interest in tag'. 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.

---

[1] `https://github.com/luisenp/sigmod14`
[2] `http://www.cs.albany.edu/~sigmod14contest/leaders.html` under team name 'shparg'

18	tow	0.740	9.843	411.609	N/A	Mar 29 - 02:08am
19	Palios	1.239	9.282	N/A	N/A	Apr 09 - 11:21pm
20	parallel_while	0.773	14.018	N/A	N/A	Apr 15 - 07:52am
21	testat	0.822	19.516	N/A	N/A	Apr 15 - 06:56am
22	shparg	4.363	30.462	N/A	N/A	Apr 15 - 06:02pm
23	HuangDuDu (University of Toronto)	6.242	134.647	N/A	N/A	Apr 11 - 06:18pm
24	Allenbj	1.324	567.282	N/A	N/A	Mar 12 - 12:27pm
25	Turtle (University of Ioannina)	14.627	N/A	N/A	N/A	Apr 15 - 10:56am

Figure 1: Final position of our team 'shparg' in the SIGMOD programming challenge.

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 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.

2.2 Query types

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

2.2.1 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.

2.2.2 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.

2.2.3   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 organizations 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.

2.2.4   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.

2.3   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 challenge 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[3] disk-based graph database system. We thought this system was appropriate because

---

[3]http://www.neo4j.org/

the queries in the challenge are largely path-oriented, and it seemed unlikely that all of the relevant data would fit in memory[4]. 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 initial 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, and any potential improvement Neo4j could offer in query performance would not offset the cost of populating the database. It seemed 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. So, we abandoned the embedded database approach.

3.2.2 In-Memory Design

We turned our attention to a simpler key-value approach tailored to the SIG-MOD challenge dataset. Specifically, we indexed nodes and edges via several simple in-memory arrays and hash tables, with one structure 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 nodes that need not persist through the query phase; examples include comments and forums. This information is required to persist only while the database is populated; for instance, comments nodes are required to compute the number of comments between people, but are not needed thereafter. We confirmed on the SIGMOD system that this design could hold a 100k person dataset in memory without exceeding the 15Gb of available memory.

The in-memory approach reduced the time it took to both index the data and answer the queries by several orders of magnitude, with respect to our initial Neo4j implementation. We accordingly moved forward with the in-memory design.

3.2.3 Data structures and indexes

Our Java implementation of the in-memory design defines the following data structures for each of the relevant entities in the contest data.

- **Person** stores a 32-bit integer representing the person's id, a 64-bit integer representing the person's birthday, a list of tag interests, a list of locations

---

[4]Although we later found that we could, in fact, fit the relevant data in memory.

and a hash map of known persons associated with the replies given to each of them.

- **Tag** stores a 32-bit integer representing the tag's id, a string representing the tag name, a hash set of persons interested in this tag and a hash set of persons interested in this tag through a forum membership.

- **Forum** stores a 32-bit integer representing the forum's id and a list of tags this forum is associated with.

For each entity in the social network we create an instance of the corresponding data structure. Each type of entity is stored in its own in-memory table, which is indexed by the 32-bit id. In the case of the persons table, since the datasets always include all persons ids in the rage (e.g., ids 0 to 9,999 for the 10k dataset) we used a pre-allocated array of size 100k with a one to one mapping between the id and the array index; this lead to faster data loading and query response times. For all other entity types, we used hash maps indexes.

Additionally, we stored the following relationship tables:

- **commentCreator** stores the 32-bit id of the person who created each comment, indexed by a 32-bit comment id. It is essentially and array where index i stores the id of the person that created the comment with id i. It's pre-allocated to a size of 700,000,000 comments, which is enough for the 100k dataset used in the SIGMOD preliminary evaluation.

- **placeOrg** is a hash map that stores the place a given organization is located at.

- **placeLocatedAtPlace** is a hash map that stores the place a given place is located at (e.g., Amherst is located in Massachusetts).

- **namePlaces** is a hash map that stores the ids of all places with a given name. Note that different places (i.e., with different ids) can have the same name string.

These tables are populated by reading files such as person.csv, comments.csv, tags.csv, comment_hasCreator_person.csv and so forth. Most of the memory consumption is due to the `commentCreator` table. However, this table is only used temporarily during data loading, to obtain the number of replies between people. The table `commentCreator` is cleared after these numbers are stored in the persons table so that freed memory can be used for other parts of the dataset.

3.3   Query Implementations

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

3.3.1 Query1

We solve Query 1 via a bidirectional breadth first search (BFS) of the person-knows-person graph based on the constraint on number of replies x. The hash map of neighbors stored by each person instance can be used to quickly prune edges that don't satisfy the constraint.

3.3.2 Query2

First, we 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. To compute the connected components, we use the list of persons that are interested in the tag, and information about the birthday stored by each person node. We use this information to create the induced graph on-the-fly and then compute the size of the largest connected component using several BFS.

3.3.3 Query3

`namePlaces` is used to find all places with the given name. For each of these places p, we perform 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 `placeLocatedAtPlace` 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 we use the hash set of persons interested in each tag (where the hash is given by person id). Therefore the similarity between two persons can be computed in linear time in the number of interest tags.

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` instance.

The induced graph the centrality score of each person is computed 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.

In our final submission we used an approximate version of this algorithm in which person nodes are first sorted by non-increasing node degree in the induced graph. Then we only considered the top 25% of the nodes as candidates for top k centrality nodes. This worked well on both the 1k and 10k datasets and reduced computation time significantly (see Figure 4). However, there are no guarantees of correctness.

In a subsequent version (developed after the SIGMOD deadline), we sorted persons by the number of persons they can reach within at most two steps, and considered only the top 5%. This worked very well on the 10k dataset, significantly reducing the time it takes to solve Query 4. However, it did not work on the 1k dataset. Nevertheless, it should be possible to devise an exact version of this strategy based on some incremental pruning of approximate centrality computations.

3.4 Concurrency

To take advantage of the eight cores provided by the SIGMOD system, we developed a multi-threaded implementation of the query solver. Since the challenge involves answering hundreds of queries, the easiest (and possibly best) way to take advantage of parallelism is to cleverly distribute queries between different threads, instead of developing complex multi-threaded implementations of each query.

In our final design, queries are uniformly distributed between eight threads, under the assumption that all queries take approximately the same time to compute. This is true for our last approximate implementation of Query 4, although not for previous implementations. In previous versions, type 4 queries were orders of magnitude more costly than the other query types. Thus, we also developed some versions in which two threads are exclusively dedicated to answer type 4 queries and the remaining threads to other query types. It is worth mentioning that we also experimented with a multi-threaded version of Query 4, although we didn't obtain any significant computational savings from this version.

4 Results

Our implementation's performance is shown in Figure 4. All times were provided by the SIGMOD submission system (measured in seconds). According to the challenge description[5], 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

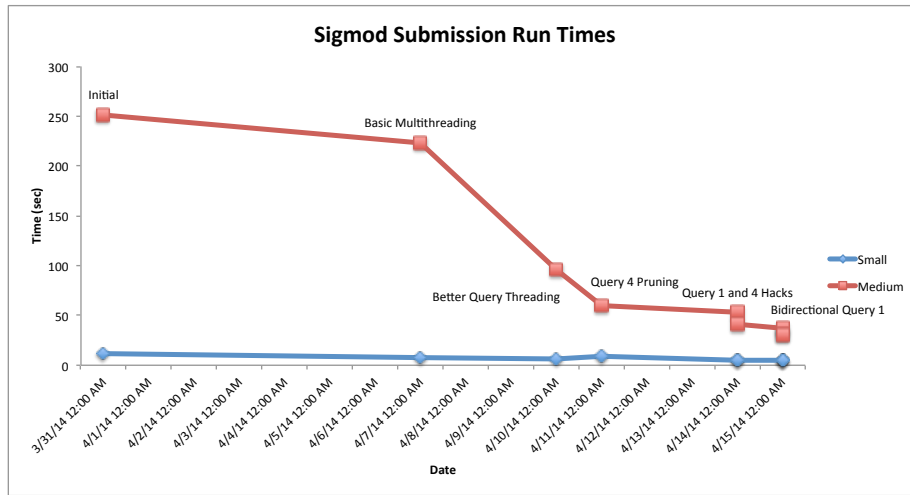[5]http://www.cs.albany.edu/~sigmod14contest/task.html

Figure 2: 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.

5 Discussion

Our implementation was about an order of magnitude slower than the best submission to SIGMOD. We speculate that we could have increased performance as follows:

5.1 Improved Concurrency

We did not introduce any concurrency into the process of loading the data. This could have improved performance significantly given that large portion of running time that loading took. Furthermore, the computation of some queries could have been initiated before all data was loaded.

5.2 Further Investigation of Heuristics

We found that the introduction of a simple heuristic to the query four implementation significantly improved the runtime of the entire pipeline; namely, sorting person by non-increasing node degree in the then we only considered the top 25%. After the contest, we found that sorting persons by two-step reachability and reducing the considered persons to the top 5% yielded additional savings. While these heuristics did not guarantee correctness, they were correct in practice. The introduction of other simple heuristics or approximate algorithms for computing graph measures could have provided more benefit.

5.3 Implementation Language

We found that performance increased when we avoided 'boxed' Java types; that is, fundamental types (like int) performed better than objects (like Integer). This suggests that a lower-level language, such as C, could have yielded performance benefits.

6 Conclusion

We developed a simple in-memory approach for performing the graph-related queries of the SIGMOD challenge. We found that more complex representations, such as an embedded graph database, performed quite poorly when the time to load and index the data is taken into consideration. Given the challenge constraints, a simple in-memory approach based on arrays and hash indices offered reasonable performance for the full challenge pipeline, and our submission was (preliminarily) ranked 22nd out of 32 total submissions.

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.