

Report Assignment 1 - Large Scale Data Engineering

Alberto Simioni, 2577392

Federico Ziliotto, 2577394

February 16, 2016

1 Data Size Analysis

The main bottleneck of the basic algorithm is memory. With only 1GB of available memory and binary files to load for a total of 5.6GB (`person`, `interest` and `knows` maps) we first analyzed the query to look for ways to drop unnecessary data. The possible optimization we found were:

1. Consider only relationships between people in the same location: the query searches for friendships of people that live in the same city, while the `knows` file contains all relationships. By removing those between people in different cities we both reduce the size of the data we have to look into and the computation cost of searching through friendships that are not important for the results of the query;
2. Consider only mutual friendships: each person can have a `knows` relationship with each other but this is unilateral. Since the query asks for only mutual relationships between two person (if `P1`, `P2` are two persons, then in the `knows` map we can find both `P1->P2` and `P2->P1`) we drop all single relationships between people;
3. Remove people that don't have any mutual friendship with someone in the same city: it follows directly from the steps above, these type of people are not useful because they will surely not be result of the query.

After applying these optimizations we created the files:

- `knows_location`: original `knows` that contains only relationships between people in the same location;
- `person_location`: `person` map with updated indexes `knows_first` and `knows_n` to reflect the `knows_location`;
- `knows_mutual`: `knows` relationships only between people that know each other;
- `person_mutual`: `person` map that contains only people that have at least a mutual friendship in the same location and has the correct indexes `knows_first` and `knows_n` (for the `knows_mutual`);

2 Interests inverted table

3 Birthday ordering

In the naive implementation we run through the entire `person_map` and check every time if the current person has the birthday in the range of the query. We decided to optimize the access to the data by ordering the person by birthday. We had two options: order the people in the data structure (and update the indexes contained in the interests and knows files) or create a new index containing the position of the person and its birthday and then order it. We decided to go for the latter that was easier to implement without modifying the other data structures but has the disadvantage that it uses more memory.

In the cruncher, we use the birthday index to first search the first occurrence of a person with a birthday in the range required by the current query, then we know all the people inside the range follows the first one until we reach someone with a birthday that is out of the bound.

4 Data Structures

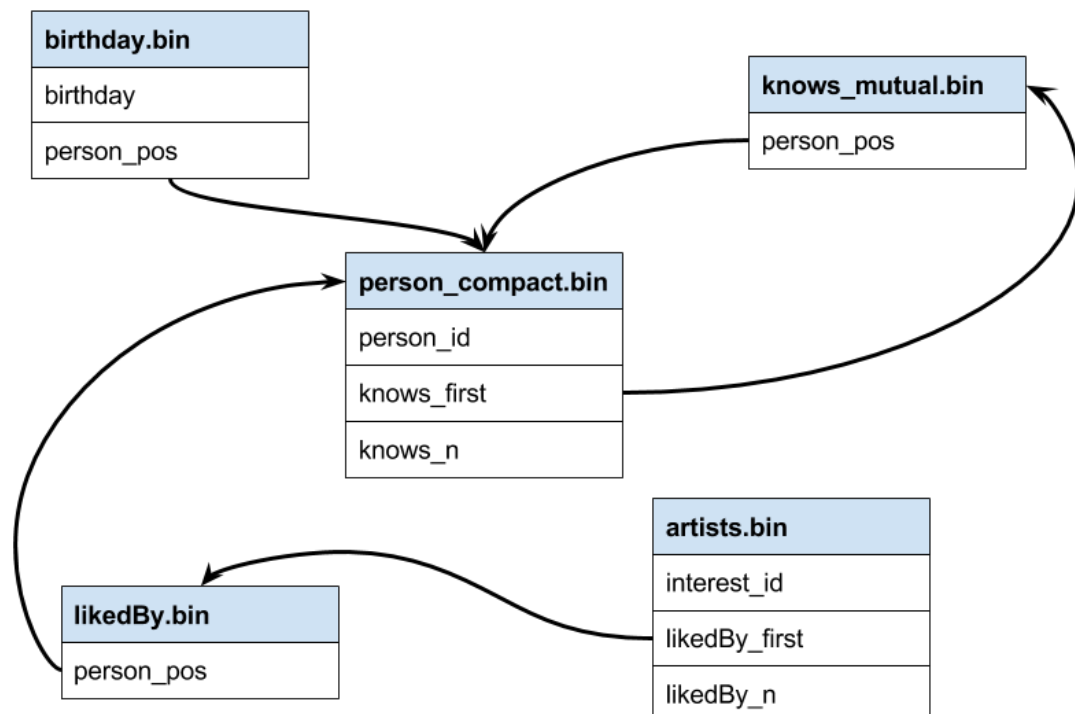


Figure 1: Data Structures

In figure 1 we show the data structure used by the cruncher to perform the queries. The intermediate files created are not shown in the figure. Since we use a different table for the people's birthdays and the interests table now has a reference to the person's one, we decided to drop the birthday, interest_first and interest_n fields from the Person data type to reduce even more the total size.

5 Cruncher

6 Possible improvements

- To save the relationships between people we use a person table that contains all the people and a knows table that has the offsets to the person table. Another way to obtain the same result would be to use a unified table of couples with two person_id for each entry. This would improve the sequential access to the data (the id of two person who are friend are always close to each other)
- Instead of using a secondary file to index the birthdays we could have ordered the people in the data structures to reduce the memory usage;
- For each query, we load the `artists` and `likedBy` files to calculate the scores for the query in questions. We could calculate the scores for all the queries (or for a bunch of them) at a time and store them in an array (the number of queries should be small enough to avoid occupying too much memory with this array) so that we wouldn't need to access those files frequently;
- An alternative to the `aartists` and `likedBy` files we use would be to create a bit array of size `number_of_person * number_of_interests` where each bit set to 1 if a person likes an artist. In a scenario where the amount of 1s is around half of the total (each person on average likes half of the artists) this would be a good way to compact the data. In this case we found that each person likes only a few of the artists available, so implementing the bit array would have led to a structure filled with mostly 0s.