On Several Social Network Analysis Problems: a Report

George Chernishev, Vsevolod Sevostyanov, Kirill Smirnov, and Ilya Shkuratov

Saint-Petersburg University, Russia
chernishev@gmail.com
vsevost@gmail.com
kirill.k.smirnov@math.spbu.ru
shkuratov.ilya@gmail.com

Abstract In this paper we describe our approach to several problems offered at the ACM SIGMOD Programming Contest 2014. These problems belong to the area of a social network analysis and involve several types of queries to a social graph. The considered graph is modeled by the standard SNB benchmark. We briefly introduce this benchmark, the contest and the problems. Next, we describe our contribution, which is the following: the algorithms for evaluation of these queries and their efficient implementation. Furthermore, we present parallelization techniques for these algorithms and describe overall architecture of our solution.

1 Introduction and Related Work

In this paper we study several problems offered at the ACM SIGMOD Programming Contest 2014 [1], a yearly programming contest focused on data management topics. This contest has a number of features, which distinguish it from a well-known ICPC series:

- Participants are offered some science-intensive task, which is usually an unsolved problem of current importance.
- The contest runs for several months and no on-site participation is required.
- Topic specificity — the clear data management focus is present. For example, contests of previous years involved construction of distributed query processing engine (2010), multidimensional index (2012) or document stream filtering system (2013).
- The participation is allowed to both graduate and undergraduate students, without any restriction on a number of attempts.

While this contest is not so well known as the ICPC, it is nevertheless popular. For example, last year there were more than 100 registered teams. The contest is relatively young — it runs for 6th time this year.

In this paper we also describe the contest: the rules, the task, its timeline and required qualifications. Moreover, we present our experiences and provide a solution of the team “GenericPeople” (Ilya Shkuratov and Vsevolod Sevostyanov), which was ranked 17 out of 33 teams on the preliminary (public) tests. While our approach is not the best, it still has merit:

our solution can serve as an example demonstrating the required qualifications and which may help to assess the required effort and work intensity. These factors may be of interest for a person who is thinking about the participation;

- the solution successfully passed through all available tests (datasets of three different sizes) within the time limits specified by the contest organizers (5 and 10 minutes);
- the proposed algorithms passed all correctness tests;
- parallelization techniques of these algorithms may be of interest;
- the number reported in the leaderboard is the sum over all query types, at the present time we can say nothing regarding their individual performance;
- at last, the number was reported for three datasets; the proposed algorithms may behave differently (better or worse) on another dataset.

Thus, we deem current study as worthy to be presented and of some interest for the reader. Another motivation for this paper is the concise presentation of the solution for the contest problem, which is usually lacking. After the contest all what is left are the posters of the top five performing teams without detailed explanation (it is given orally at the conference). Also, these posters are (or at least were in the past years) not going into the conference proceedings and are kept on a website, which may disappear. Moreover, we present our experiences and describe (at least partially) the way we went through in order to produce a working solution. It is impossible to pass on all these aspects via poster.

This year contest was dedicated to a social network analysis topic. Social network is essentially a graph, whose vertices represent users and edges denote relations between them. An example of such relation may be “know each other”, “follow” and so on. Additionally extra information like a place of work or study, geographical information, various tags, images, likes etc. is known.

In the past years massive amounts of such information were made available for analysis, forming a strong incentive for both academy and industry to come with means for its efficient storage and processing. Social data play a significant role in the whole “Big Data” movement.

A lot of analysis tools employ the MapReduce \textsuperscript{6} programming model. Industrial examples of such systems are PIG (Yahoo) \textsuperscript{13}, SCOPE (Microsoft) \textsuperscript{5}, Hive (Facebook) \textsuperscript{18}, Dremel (Google) \textsuperscript{14}. Academic examples are Starfish \textsuperscript{9}, HadoopDB \textsuperscript{3} and many others\textsuperscript{2}. An alternative (which can be considered a poor man’s solution) sometimes employed in production environment, is to use scripts written in scripting language like Python to commence the analysis. A data scientist has to analyze the problem and implement all necessary algorithms manually. While it may not favor the rapid development, it may allow to achieve a more efficient processing. Naturally, this approach is more flexible than using a standard tool and allows a fine-tuning of algorithms. However, it requires extensive technical expertise: knowledge of algorithms and data structures, the understanding of the data processing and so on. The tasks of the contest are representative examples of this “manual” approach and can be considered as a training for a data scientist.

\textsuperscript{2} A list can be found in http://dl.acm.org/citation.cfm?id=1454166, last accessed 22/07/2014.
Another aspect of the contest task is the graph analysis component. Graph analysis is a mature area of research which studies the efficient storage and processing of graph data. There are several graph database management systems (a special type of DBMS) and graph programming frameworks. These DBMS feature special query languages, query processing algorithms and data storage.

Some examples of the graph DBMS are Neo4j [12], InfiniteGraph [10] and the framework examples are Apache Giraph [2], Signal/Collect [17]. It is necessary to mention that two latter systems also follow the MapReduce model.

The contestants were given the task which consists of the datasets and four types of queries. The social graph was generated using the SNB [16] tool.

The goal was to develop a program which computes the results as fast as possible. The contestants had not only to devise the algorithms for efficient query processing on a large graph, but also to parallelize them. This is a must, given the fact that the evaluation of the resulting implementation was performed on a server-class equipment (8 cores).

Another important aspect was the order of computation for each sub-query. The contestants had to bear in mind the size of intermediate results and the memory bound. In other words, the contestants had to perform the work of a query optimizer: gather needed statistics, assess selectivities and develop an optimal processing strategy for each query type. Also, given the hardware multi-core capability, efficient inter-query type orders are also of interest.

The contribution of this paper is the following:

– The description of the ACM SIGMOD Programming Contest 2014 and its task;
– The contest from the participant’s point of view: our experiences;
– The algorithms to handle the problems offered at the contest;
– A parallelization techniques for each of these algorithms;
– A general system architecture: subquery computation orders, inter-query type orders and chunk-based data loading.

Now, we are going to describe our experiences. The task description, SNB description and its data schema, the data statistics are presented the appendix section.

2 Contest description and experiences

Let’s describe this year contest from the participants’ point of view. We have already briefly described the contest and its specifics in the introduction section. You can find detailed information regarding the ACM SIGMOD Programming Contest series in the reference [19].

Our research group is a frequent participant of this contest; we had achieved good results twice in the past: in the 2010 [3] (team “spbu”) and 2013 [4] (team “Rota Fortunae”) year. Both times our teams achieved 3rd place in the final ranking.

\[\text{http://dbweb.enst.fr/events/sigmod10contest/results/#winner},\quad \text{last accessed 22/07/2014.}\]

\[\text{http://sigmod.kaust.edu.sa/finalists.html},\quad \text{last accessed 22/07/2014.}\]
2.1 General information

This year contest followed the general scheme described in the reference [19]. However, there were several notable divergences:

1. The contest started noticeable later compared to previous years;
2. There were no 2\textsuperscript{nd} round, unlike early years. This change happened in 2013;
3. The absence of the dedicated correctness testing phase during the evaluation (it was performed concurrently with the performance evaluation);
4. There was a series of datasets which were progressively disclosed by the organizers, as the performance of the submissions improved;
5. The task did not explicitly required parallelization or concurrency support, but instead, implied it. It was possible to submit purely sequential implementation;
6. It was possible to submit only the executable, without source code during the preliminary evaluation. The final evaluation required source code and this led to some compatibility difficulties;
7. Contestants were allowed to choose programming languages other than C++.

The provided task was a science-oriented problem related to social network analysis. The problem was to execute a number of queries to a graph representing some social network. The goal was to produce a correct answer and minimize the overall processing time. The graph and queries are fully described in the next section.

Below you can see the timeline of the contest.

- January 25, 2014 — Contest announced.
- February 1, 2014 — Detailed specification of the requirements and test data available.
- February 16, 2014 — A medium data set (10k people) with query workload and answers are available on the Task page. New query workload and answers for the small data set (1k people) are available on the Task page.
- March 1, 2014 — Team registration begins. Leaderboard available.
- March 11, 2014 — Workloads on a medium data set (10k people) have been added to the evaluation system.
- March 17, 2014 — Workloads on a large data set (100k people) have been added to the evaluation system.
- April 15, 2014 — Final submission deadline.

In the overall the contest run for two and a half months. Also you can see that several datasets were progressively added to the evaluation pool. These datasets were progressively disclosed by the organizers as the performance of submissions improved. This is a rather new model of evaluation (appeared in 2013 contest) and it was employed in the following way. As soon as the several submissions were achieving some performance level, where it was hard to discern their quality due to inaccurate measurements (thread scheduling effects, for example), a new, larger dataset was added.
2.2 Communication with the contest organizers

Information about the order and rules of the contest were provided on a special web page [1], which was the main mean of communication between the organizers and the contestants. It also describes test data sets, the task and an evaluation environment. Later opportunities to register a team and submit solutions were added.

The organizers also created a Google Group in order to discuss any technical issues (e.g. code page problems) and to provide additional information that might be of interest to all of the contestants: test datasets publication dates, disk space availability, size of data set for the final evaluation and so on.

2.3 Required skills and our experiences

Since the organizers of the contest considers Linux as its target platform, we decided to use C++ programming language as it looks to us an highly-optimizable one. Those who want to take part in the contest are advised to learn Linux development utilities such as gcc, make, valgrind (especially callgrind might be useful), gdb, etc. Also two bash scripts were required: one should build the solution and the other — run it with certain parameters.

You also may encounter restriction on size of submitted solution. It was 8 MB this year, thereby it was helpful for us to learn a couple of gcc flags. The first one is -s. It removes unneeded symbols from an executable, thus reducing its size without the loss of performance. The second flag may be useful, if you use external libraries: -MM instructs the compiler to generate source files dependencies. This helped us to familiarize with boost headers dependencies, strip boost from unneeded header files and further reduce submitted archive size.

Understanding compiler optimization methods may be of use as well. It allowed us to cope with the gcc optimizer bug, namely incorrect copy propagation after global common subexpression elimination pass. It leads to usage of the original pointer to the buffer instead of its copy, which cause segmentation fault on an attempt to free this buffer. The workaround is to add a dummy use of the original pointer after working with the buffer.

Another important skill is an ability to find necessary information on the subjects of the competition, i.e. the ability to work with digital libraries. Usually the task of the competition (or one of the tasks) is an unsolved scientific problem. Thus one may find useful information about methods have been tried or perspective approaches. These gave us several hints for the given task.

2.4 Tools

Aside from the usual requirements this year contest posed an additional one: knowledge of some scripting language or a tool for data analysis. This language can be used for datamining: to detect hidden dependencies in the source data and collect necessary statistics. We used python programming language, other examples include R and Octave tools.
2.5 Data

The schema for the data used in the task formulation is presented on Figure 1. Data were stored as a set of CSV files. It is worthy to mention that not all of the files were needed for the query processing. Also, organizers had provided data only for two datasets — the one containing thousand and the one containing ten thousand of persons. These datasets are sufficient for the debug purposes, but they are not enough to tune algorithms for the final evaluation, which involved a graph of million of persons. The benchmark generation parameters were kept in secret and it was impossible to generate that graph by ourselves.

3 Algorithms

In the rest of this paper we refer to the graph induced by “know each other” relation as graph, and to the breadth-first search of that graph as BFS. This graph is used in every query type and BFS (as we show further) plays the key role in all of them. Thus, a shorthand notation would be useful. The rest of notation can be found in Table 1.

3.1 Query Type 1 (Shortest Distance Over Frequent Communication Paths)

Algorithm description An obvious strategy for evaluation of such query would be the following:

1. Run BFS from person p1 to person p2 and return hops count;
2. During the BFS traversal one needs to check the replies condition. For each edge, considered on a given BFS step, one has to calculate the number of mutual replies for the corresponding persons. If it is less than k, then the transition is not possible — the edge does not exist.

This “naive” approach needs no preparation and can be ran just after the graph construction. For each pair of adjacent persons it is necessary to calculate the number of replies and this may take some time. Thus, the described BFS has the complexity $O(m \cdot n \cdot (|V| + |E|))$ where $n$ denotes a cardinality of “comment is reply of comment” relation and $m$ — cardinality of “comment has creator person”.

Therefore, we propose a pretreatment phase that will compute number of replies once, which effectively eliminates the repeated calculations. Our goal is to find persons that made not less than $k$ comments replying to each other. For each pair of persons connected by an edge $e$ in the graph we will determine the number of mutual replies $k_e$ and attribute it to $e$. In this way, BFS on each step compares two numbers: given $k$ and pre-calculated $k_e$.

Algorithm details Data preparation. We transform relation “comment is reply of comment” into “person replied to person” using “comment has creator person” and then count number of replies for each element in the resulting relation.

First, file “comment has creator person” is parsed and represented as map $ccreat : comment \rightarrow creator$ in order to reduce comment owner search time. Second, since the file representing “comment is reply of comment” is one of the largest generated by
the SNB it is of use to distribute its processing between separate threads. Each thread calculates a local result $\lambda$ and then all local results are merged into replies map

$$\Lambda : (\text{person}_1, \text{person}_2) \mapsto$$

$$\mapsto (\text{replies of person}_1 \text{ to person}_2,$$

$$\text{replies of person}_2 \text{ to person}_1).$$

Next, from the replies pair minimal coordinate is chosen (we previously denoted that value $k_e$) and ascribed to the edge between $\text{person}_1$ and $\text{person}_2$. For those persons who hasn’t written any replies to each other, $k_e$ is set to 0. The proposed data preparation procedure has the complexity $O(m + n + |E|)$.

**BFS.** Now we have sufficient information to tell whether a pair of persons has at least $k$ mutual replies. With this information at hand, BFS makes only one numeric comparison per hop. Therefore, BFS complexity is reduced to $O(|V| + |E|)$.

### Algorithm 1: Local Replies Processing

1. function $q_1\text{\_local}(\text{crepl}, \text{ccreat})$
2. input: $\text{crepl}$ — part of file comment is reply of comment,
   $\text{ccreat}$ — map comment $\leftrightarrow$ creator
3. output: $\lambda$ — local result, same type as $\Lambda$
4. $\text{reply\_ID} \leftarrow \text{read\_next}(\text{crepl})$;
5. $\text{comm\_ID} \leftarrow \text{read\_next}(\text{crepl})$;
6. while not eof($\text{crepl}$) do
7.   $r\_owner \leftarrow \text{ccreat} [\text{reply\_ID}]$;
8.   $c\_owner \leftarrow \text{ccreat} [\text{comm\_ID}]$;
9.   $\text{first} \leftarrow \min(r\_owner, c\_owner)$;
10. $\text{second} \leftarrow \text{other}$;
11. if $\exists(\text{first}, \text{second}) \text{ in } \lambda$ then
12.   increment coordinate in $\lambda [\text{first, second}]$ according to reply direction;
13. else
14.  $\lambda \leftarrow (\text{first}, \text{second}) \mapsto (1, 0) \text{ or } (0, 1)$ according to reply direction;
15. return $\lambda$;

3.2 Query Type 2 (Interests with Large Communities)

In order to efficiently evaluate this query type one has to efficiently perform the search for connected components on a graph.

A general idea This query type is evaluated using the graph which is derived from the graph by applying two additional restrictions. The first one states that all vertices should share a common tag and the second one requires the birthdate of every vertex (person) to be later or equal than a date $d$. In this filtered graph we should search for connected components. The straightforward approach is to check these restrictions during the graph traversal. This means we should traverse the whole graph for each tag. The complexity
Algorithm 2: Global Replies Processing

```plaintext
function q1_global(graph, crepl, ccreat, threads_number)
    input: graph,
           crepl — file comment is reply of comment,
           ccreat — map comment → creator,
           threads_number
    array p ← split crepl into threads_number parts;
    for j ← 0 to threads_number do
        create_thread(q1_local(p[j], ccreat));
    for j ← 0 to threads_number do
        merge(Λ, join_thread(j));
    for e in graph do
        p1 ← e.start();
        p2 ← e.end();
        k_e ← min(Λ[(p1, p2)].f, Λ[(p1, p2)].s);
```

of this approach is $O((|V| + |E|) \cdot |T|)$. Now, let's consider other approaches to restriction checking.

**The common tag requirement.** In order to handle this restriction an index tag-person using the person_hasInterest_tag relation can be built. One should not worry about hitting the memory bound. Table 3 shows that the number of tags is increasing slowly, on average a person has 3.5 interest tags in graph containing 1000 persons, as well as in graph containing 10000 persons. Thus, we assume this ration will stay the same during the further graph size increase. Using these numbers we can estimate the size of the index for a graph containing million persons: $((4500 \cdot 9) + (1000000 \cdot 3.5)) \cdot (8/1024/1024) \approx 27$MB.

This index is represented by a hash-table (implemented by boost::unordered_map collection). It allows to efficiently construct and query the index, having constant insert and access complexity \cite{4}.

**Birthdate.** We can reduce the number of constraint checks by pre-filtering persons (vertices) who were born earlier than a given date. These excessive checks happen during the processing of arcs which lead to the aforementioned vertices. Thus, it may be impractical to construct the whole new graph. Indeed, suppose we are given a person which has ten friends which share the common tag. In this case when we will check this vertex eleven times (adding one more time for index look-up during BFS initiation). But if we properly exclude it from the neighboors of these ten nodes, we would have to check it only once.

Thus, in order to accomplish the pre-filtering, we would exclude not the vertices but the edges incident to them. We call this process “date pre-filtering” and the resulting graph — “graph time slice”, or simply “time slice”. This approach allows us to keep arrays of such edges in our graph. The resulting increase of the memory requirements is tolerable; this form of storage would require not more than two times the space of the original graph. It is acceptable even for a graph with a million vertices.
However, this approach requires a preprocessing phase because every query would require its own time slice, which would lead to additional costs synchronization (in case of multithreaded approach). If we would manage to avoid preprocessing phase, we would be able to run several queries consequently and without synchronization. This will lead to a better load balancing because the thread, which had finished its part of work for a previous query, can immediately start processing next query without waiting for others. We will refer to this type of processing as “continuous processing”.

It is impossible to create all of the time slices to avoid preprocessing due to memory bound. But it is possible to create several of them and later use the one which contains $d$. If the time slices form the decomposition of the whole graph time range (complete, pairwise disjoint) $G_t$ and the graph contains million persons, this approach would additionally require $22.8 \cdot (N - 1)$ MBs (compared to a single time slice), where $N$ is the number of time slices. Thus, we managed to avoid preprocessing phase using rather small amount of memory and sacrificing some accuracy of the time slice.

**Query processing** Index construction. The index is constructed using the file `person_hasInterest_tag`, which defines the corresponding relation. This file is being parsed and pairs (tag_id, person_id) are being inserted into hash-table, with the key being tag_id, and the value — array of person_id.

The index is being constructed using one thread because the file `person_hasInterest_tag` is rather small and there would be a considerable overhead which is not covered by the gain (for a large datasets it is possible that parallelization will pay off).

**Time slice construction.** We had studied the distribution of the persons over time (see Table 2) and suggested that there is a uniform distribution. Also, the contest organizers stated nothing regarding the query parameters. Thus, we considered $d$ as a uniformly distributed random variable. On that basis we decided to produce time slices of equal length.

There are two stages of time slices construction:

- On the first stage we determine the time slice which should contain a given person. In all of the datasets the whole time range was eleven years (1980 – 1990 years). In order to keep the balance between time slice accuracy and memory expenses, we decided to split the whole range in six slices: five of them span two years and the last slice — one year.
- On the second stage the time slices are constructed (see Listing 3). It is easily parallelizable because all of the modifications which are performed in the graph vertices are independent of each other.

**Community cardinality computation** Let’s first describe the sequential algorithm and then discuss its parallelization.

**Sequential algorithm.** Before we start to loop over tags, we initialize a structure for the result by specifying the maximum number of tags it should keep (line 2), then we scan all tags (lines 3–9). For each tag we get the cardinality of the largest connected component using the BFS (lines 4–8) and put it into the resulting list (line 9).
Algorithm 3: Time Slice Construction

1 function slice(graph)
   input: social network graph graph
   foreach person in graph do
      foreach friend of person do
         /* insert friend to person’s slices array that corresponds to friend’s slice id */
         person:slices[friend:slice_id].add(friend)

Algorithm 4: Find Top-k Interest with Large Communities. Sequential.

1 function query2_linear(tp_index, query)
   input: tag — person index tp_index, query parameters query
   output: top-k list
   top_k_tags.init(query,k);
   foreach (tag, person) in tp_index do
      component_size ← 0;
      foreach person in Persons do
         if person is visited then
            new_size ← BFS(person, query.date);
            component_size ← max(component_size, new_size);
            top_k_tags.insert(component_size, tag);
      return top_k_tags.get_top_k_tags();

Parallel algorithm. We tried to minimize the overheads of synchronization when we were parallelizing the algorithms. In order to achieve this one has to minimize the amount of shared resources. In our case this shared resource is the top_k_list, which we decided to keep as a thread-local top_k_list. Threads construct a partial results independently and then one merges them in order to get the final result. Since we managed to avoid the preprocessing phase, the queries can be processed in blocks, e.g. without barrier synchronization between two consequent queries. Also, it is possible to delay the evaluation of the final result until the end of the query block. Basing on the amount of memory, which we are ready to dedicate for partial results, we can determine the size of the block. We assumed that the number queries of the second type would be low, so we grouped all of them in one block. We can name two advantages of this approach:

- no synchronization is needed in order to form the final result for each query;
- it is possible to parallelize the construction of the final results.

Thus, we eliminated excessive synchronization and increase the number of operations which can be executed in parallel.

Listing 5 and Listing 6 describe the parallel approach. Algorithm described in Listing 5 differs from the sequential counterpart in the tag distribution between threads (for this purpose a thread id and the total number of threads are used). Here, we assume that the threads id range from 0 to N - 1, where N is the total number of threads. In the algorithm Listing 6 we describe the getting of the partial results (line 8) and their merging (lines 8 - 9).
Algorithm 5: FIND TOP-K INTEREST WITH LARGE COMMUNITIES. PARALLEL.

function query2_parallel(query, tp_index, th_info)
  input: query parameters query,
         tag — person index tp_index,
         thread information th_info,
         (th_info.id — thread id,
         th_info.count — total number of threads)
  output: partial result for query partial_result
  int position ← th_info.id;
  partial_result.init(query.k);
  while position < tp_index.size do
    /* Get pair (key, value) from index at position */
    (tag, persons) ← tp_index.get(position);
    foreach person in persons do
      if person is visited then
        new_size ← BFS(person, query.date);
        component_size ← max(component_size, new_size);
      partial_result.insert(component_size, tag);
      position ← position + th_info.count
    return partial_result;

Algorithm 6: QUERY TYPE 2 EXECUTION

function evaluate_query2(QL2, tp_index, th_info)
  input: list of queries type 2 QL2,
         tag — person index tp_index,
         thread information th_info,
         (th_info.id — thread id,
         th_info.count — total number of threads)
  output: results for block of queries type 2 q2_results
  /* Queries evaluation */
  foreach query in QL2 do
    q2_part_results[query.num][thread.id] ← query2_parallel(query, tp_index, th_info);
  barrier;
  /* Merging the partial results */
  int position ← th_info.id;
  while position < QL2.size do
    q2_results[position] ← merge(q2_part_results[query.num]);
    position ← position + th_info.count
  return q2_results;
Possible improvements Further improvement may be achieved through tag sorting by a number of persons with this tag (storing the lower bound). While looping we can check whether there are enough persons with the given tag left in order for the result to enter the top-k.

3.3 Query Type 3 (Socialization Suggestion)

Algorithm description. The common sense may provide the following idea of the straightforward evaluation:

1. for each vertex \( v \) in the graph perform BFS while keeping in mind the given hops count \( h \);
2. upon completion BFS returns the list of reached people \( rp \);
3. for \( v \) and each person \( v_r \) from \( rp \) check information about their work places, study places and location for correlation with \( p \);
4. if one of the places where both \( v \) and \( v_r \) are involved is \( p \) or its subplace, then calculate the number of common interests \( ci \);
5. store (sorted by \( ci \)) the resulting pairs \( (v, v_r) \);
6. return the top-k pairs as a result.

This algorithm requires examination of all the persons returned by BFS. Since \( graph \) is a social its edge count follows power law, therefore there are some hubs and connectors with large degree and many vertices with only a few incident edges [11]. Hubs and connectors shorten the paths between persons and thus, the size of \( rp \) may be significant. The time complexity of this algorithm is

\[
O(|V| \cdot (|V| + |E| + |rp| \cdot |person.places| + |person.interests|)).
\]

It is desirable to reduce the number of persons to examine without the loss of result correctness. In order to do that we suggest to group persons by some of place types. SNB provides three place types: city, country and continent. There are only five continents and such a fragmentation won’t be useful in context of fast-growing \(|V|\). On the contrary, grouping by city leads to lots of small groups that take large amount of space. Thus, country seems to be a good choice which causes medium number of satisfactory-sized groups (about 111 countries for 1k vertices). Furthermore, study places and location of person are represented by cities and work places — by country, respectively. Thus, we can easily make all three place classes identical using “place is part of place” relation, provided by SNB.

But still if the type of \( p \) is continent, we will have to iterate the whole \( graph \). There are several solutions to this problem, some of them are the following:

1. for each continent store all countries that are situated on it;
2. for each person store continents which are associated with that person.

The first solution is not as space consuming as the second yet requires more time. For each person it is necessary to run BFS several times (one for each country) or for each hop to check whether any of the person’s places belongs to a given continent.
The second solution has a great advantage: for each BFS hop it takes only a few numeric comparisons to check the required condition. Every person will require at max 5 integers of additional storage, so such approach is affordable and will not dramatically increase graph representation’s size. Let’s denote BFS that returns reached persons from the given country as gBFS and BFS that returns reached persons from the given continent as cBFS.

Using the proposed partitioning we suggest a following algorithm:

1. use the type of person to determine which group to process;
2. for each person from this group perform gBFS (cBFS) bearing in mind the given hops count \( h \);
3. on completion gBFS (cBFS) returns the list of reached people \( rp \);
4. for each person \( v \) and each \( v_r \) from \( rp \) calculate the number of common interests \( ci \);
5. store (sorted by \( ci \)) resulting pairs \( (v, v_r) \);
6. return the top-\( k \) pairs as a result.

Described approach time complexity is:

\[
O(|\text{persons in } p| \cdot (|V| + |E| + |rp| \cdot |\text{person.interests}|)).
\]

Algorithm details. Relation “place is part of place” will be frequently accessed so it seems beneficial to implement it as a map \( \text{sm} : \text{place} \rightarrow \text{parent} \). Each vertex contains two sets of places: (i) study places and location, (ii) work places. We follow such partitioning scheme with regard to the place types mentioned above. Each person has a set of continents that are simply deduced from this person’s places using \( \text{sm} \). Groups by country place type are created during person-place relations parsing. We implement set of groups as a map \( \text{set of persons} : \text{place} \rightarrow \text{set of persons} \). Though elements of \( \text{set of persons} \) may intersect, it is convenient to iterate through sets and keep only unique persons in each group. Preparations have complexity \( O(n + m) \) where \( n \) stands for cardinality of “place is part of place” and \( m \) for summary cardinality of person-place relations.

The full processing scheme for query type 3 is presented in the following algorithms: \( \text{[6]} \), \( \text{[6]} \), \( \text{[6]} \) and \( \text{[6]} \). Due to the space constraints the first three of them are presented in the appendix.

3.4 Query Type 4 (Most Central People)

Index construction. In order to ensure the fast construction of the graph needed for closeness centrality computation we decided to build index tag - person, which maps each tag into a set of persons who are members of the forum having such tag. We set up this index using the standard C++ containers C++ map and set, which are implemented using the red-black tree. These containers were chosen due to the two reasons. First, this index should reside in the main memory. Second, in order to efficiently construct graph for a given tag we need sorted sequence of person ids. If we use a tree-based index, the sorting happens during the index construction. It is worthy to mention that index was built only for tags which were needed for the query execution: there were only a few queries of this type in the workloads, but the number of forums were rapidly increasing.
Algorithm 7: Query3 evaluation

```python
function query3(graph, h, k, p)
    /* top-k is a set with some priority that orders pairs by ci, it
     * only keeps not more than k elements at every moment */
    ptype ← type of p;
    if ptype is continent then
        for v in graph do
            if not v in p then continue;
            rp ← cBFS(v, p);
            for v_r in rp do
                ci ← get_ci(v, v_r);
                pair ← (v, v_r) in lexicog. order;
                top-k ← (pair, ci);
    else
        g ← Γ(p);
        for v in g do
            rp ← gBFS(v, g);
            for v_r in rp do
                if ptype is city and not v in p then continue;
                /* rp contains people from the whole country, not
                 * only p */
                ci ← get_ci(v, v_r);
                pair ← (v, v_r) in lexicog. order;
                top-k ← (pair, ci);
```

However, the relation which we are interested in — (tag, person) was not given explicitly, but was defined by two others: (forum, tag) and (forum, person). Therefore, at first, we had to create two indexes of these relations: tag - forum and forum - person. These data structures aim to decrease the time it takes to join these two tables, to decrease the construction time of tag - person, and also to decrease a graph construction time for a given tag:

- An index forum - person was represented by a hash-table, in order to achieve look-up complexity for the join operation of $O(1)$.
- A set was used for tag-person index in order to get free duplicate elimination during the join operation and to acquire sorted sequence of person ids during the construction of a graph for a given tag.

The join is represented by the block “Build index” on Figure 4.

The calculation of closeness centrality metric In order to calculate the closeness centrality metric one needs to know the following three values for each of the graph vertices:
1. \( r(p) \) — number of vertices, reachable from it;
2. \( s(p) \) — the sum of all geodesic distances of all of reachable vertices;
3. \( n \) — the number of vertices in a given graph.

We should note, that our graph is an undirected graph, therefore \( r(p) \) can be calculated once for each connected component. Thus, the problem is how to compute \( s(p) \).

**An algorithm selection.** Given the fact that our graphs is an undirected one and the edges are of unit weights, a simple BFS modification would suffice for the evaluation of \( s(p) \). For this purpose we can label each visited vertex with the distance to the initial one. In this approach we do not increase asymptotic complexity of BFS and do not use additional memory. We would require \( O(|V| + |E|) \) time and \( O(|V| + |E|) \) memory. This estimation is better than estimation for many classical algorithms oriented for general cases of problem “minimal distance from one vertex to all other”. For example, Dijkstra algorithm [7] for graphs with non-negative weights, based on Fibonacci heap [8] uses \( O(|V| + |E|) \) memory and \( O(|V| \cdot \log |V| + |E|) \) time. Moreover, our approach is easily parallelizable: we can compute \( s(p) \) in parallel for different vertices.

**The cut-off heuristic.** One can note that *closeness centrality* is inversely proportional to \( s(p) \) within a connected component. Indeed, it holds due to a fact that \( n \) is a constant for the whole graph and \( r(p) \) is a constant for all vertices in a connected component. Thus, we can propose a criterion for a vertex to enter the top-\( k \) of a given connected component which uses it’s \( s(p) \). Let’s define a threshold:

\[
\Theta = \max_{p \in \text{current}\_\text{top}\_k} s(p).
\]

Now, we can interrupt the computation of \( s(p) \), if the current value had exceeded the threshold \( \Theta \).

Despite the simplicity of this cut-off heuristics it drastically decreased the evaluation time for the fourth query type. Unfortunately, we do not know the number and parameters of queries of this type during the final evaluation. But the implementation of this heuristic allowed to decrease the evaluation time for more than 380 seconds on a graph containing 100 thousand persons. The resulting time was 220 seconds.

**An overall processing scheme.** The algorithm presented in Listing 8 provides more detailed description for computation of *closeness centrality* for a single vertex. Its inputs are the vertex for which the *closeness centrality* should be computed, the size of the graph and the number of connected component the vertex belongs to. We perform the graph traversal starting from the vertex source during which we store the distances in the array hops. Afterwards, we use them for the computation of \( s(p) \) (line 14). A cut-off heuristic is implemented in the following way: the line 15 shows the checks which examine whether the given vertex can get into top-\( k \) and the line 17 shows the update of the threshold.

Like for query type two, the query type four is eligible for delayed computation of the final result, which allows to decrease the number of synchronizations and to increase the parallelization capability of the algorithm. But also, like in case of the second query type the the amount of memory which we can dedicate for the storage of partial results, restricts the number of queries which we can process this way. The general processing chart for a set of type four queries is presented on Figure 4. It shows the major stages and synchronization points.
Algorithm 8: Compute closeness centrality for a single vertex

function BFS_q4(source, size, component)
input: start vertex source, 
      size of tag graph size, 
      id of connected component which start vertex belongs to component 
output: closeness centrality of source or -1  

    /* Initialization */  
    Q ← source.id;  // Queue  
    rp ← 0;  // r(p)  
    sp ← 0;  // s(p)  
    hops ← array of -1 that size is size;  // distances from source  

    /* BFS */  
    Enqueue(Q, source);  
    hops[source.id] ← 0;  

    while Q ≠ ∅ do  
        person ← Dequeue(Q);  
        foreach friend of person do  
            if hops[friend.id] == -1 then  
                Enqueue(Q, friend);  
                // set distance from source  
                hops[friend.id] ← hops[person.id] + 1;  
                rp ← rp + 1;  
                sp ← sp + hops[friend.id];  
                if sp > get_threshold(component) then  
                    return -1;  
            
        update_threshold(sp, component);  
    return rp · rp / ((size − 1) · sp);  

The first stage was described earlier, stages 2 and 6 are trivial, stage 7 is analogous to the merge of results during the processing of the second query type. Stage 5 is the reiteration of the algorithm presented in Listing 8 for all vertices of the constructed graph. Here, each thread processes its own set of vertices. The stage 4 is the preprocessing step and is committed by one thread. On this stage the vertices are marked by the numbers their corresponding connected components, the cardinality of these components is also evaluated and initialization of data structures required for further processing is performed.

The stage number 3 constructs the graph for a given tag. The graph is represented by an array of vertices, where each vertex keeps an array of neighbor vertices. The ids of persons corresponding to each tag are distributed between threads and each thread creates a list of friends for a given person in a new graph. The only ones left are the persons who correspond to a given tag in index.

Other approaches. In the last few days of the contest we found the solution that fits almost perfectly into the described problem [15]. It is developed for directed graphs with non-negative weights and reuses the CCV of a single vertex in order to estimate CCV for other vertices and reduce the further computations. Authors also use estimates
in order to produce the cut-off of vertices which not to get into top-k. That method could be modified to take into account the memory restrictions. The experiments described by authors show that this approach may be particularly efficient for unweighted, undirected graph of a large size. It can reduce the amount of computations for a majority of vertices or even avoid their processing at all.

4 System architecture

Graph structure. Considering the graph structure we bear in mind the following: (i) the cardinality of vertices may run up to a million, (ii) BFS is crucial for the evaluation of every query type. Therefore, our approach must have low memory footprint and provide efficient BFS evaluation. In order to satisfy these requirements we use representation similar to adjacency lists, but with arrays instead, that is, each vertex contains a pointer to an array of adjacent vertices. It allows us to meet the memory constrains and avoid unnecessary comparisons in the BFS implementation.

Layers. Three layers may be distinguished in our implementation: (i) file loading, (ii) structure initialization and preparation, (iii) query evaluation.

This layered structure is rather natural to the task and allows some flexibility in the setting up the order of query evaluation. That is a rather important feature for the performance improvement. The use of the first layer is to provide the interface to chunk-based file loading. It copes with the problem of big files which can be up several gigabytes in size. The use of the second layer is to parse loaded files and to build indexes and other structures required for the query evaluation. The last layer is responsible for the final results formation.

5 Experiments

In this paper we present some experiments illustrating the performance of our approach. Unfortunately, we could not provide detailed experimental data from the contest due to several reasons: (i) we do not have access to the final benchmarks (they are not yet released to the public); (ii) we no more have access to the hardware used for the evaluation by the organizers (it was a server-class one); (iii) the two largest benchmarking query sets are unavailable too (we used the largest available dataset — the medium dataset, containing 10k persons).

Thus, we had to perform experiments on our own. The hardware and software setup was the following: i7-4930K CPU (6 cores), P9 X79WS motherboard, 4GB RAM; Ubuntu 14.04, kernel 3.13.0-24, x86_64.

The first series of experiments is presented on Figure 2. They illustrate the basic approach when we sequentially evaluate queries of the same type. The results show the contribution of each query type to the overall processing time. In this series we vary the number of threads. Eventually we get a U-shaped graph, which shows that it’s not useful to employ more than four threads for the processing in this scenario. It is the result of the algorithm parallelization imperfection (not all algorithms use all cores all the time) and of the synchronization overheads. This leads us to the idea of pre-treatment phase
which will allow us to balance the load. The load balancing will be done by grouping
tasks together into stages and reordering of query types.

To examine our idea, we had split the query evaluation into the following stages:
(i) Q3 evaluation and Q1 preparation part 1, (ii) Q1 preparation part 2, Q2 preparation
and Q4 preparation, (iii) Q1 evaluation, (iv) Q2 evaluation, (v) Q4 evaluation. Tasks
belonging to one stage are executed in parallel. Figure 3 shows the results for this kind
of processing. Despite that in fact we used our idea in the first two stages only, the
performance boost of the evaluation with six threads is about 28% (compared to the best
performance from Figure 2) and 56% comparing the performance with the six threads.
This may be considered a good result for the medium dataset, which we use for testing.
Efficiency of such task grouping is determined by the “closeness” of tasks executed in
parallel in terms of time. The closer times of execution, the more efficiently we use the
processor. We can perform the load balancing in two ways: by varying the number of
threads for one task and by varying the number of tasks. Hence we can use this approach
to tune performance further. However, effect of the load balancing may vary with the
dataset. Taking such variation into account is rather difficult and requires a more detailed
study of the data structures and the algorithms involved.

6 Conclusions

In this paper we described the ACM SIGMOD Contest 2014, its tasks, timeline and our
experiences. Also we presented our approach to the offered problems and described the
advantages over the naive processing. We discussed algorithms as well as parallelization
techniques and presented the general system architecture. Its key points are the follow-
ing: query type intermixing, query type reordering, continuous query processing and
block file loading techniques.

References

3. Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander
Rasin. 2009. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for
duction to Algorithms (2nd ed.). MIT Press, Cambridge, MA, USA.
5. Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver,
and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets.
Proc. VLDB Endow. 1, 2 (August 2008), 1265–1276.
7. E. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs”, Numerische math-
Appendix: Problems

The contest offered the following problems (we fully provide them here for the better understanding of the reader and in case of the original web site outage):

1. **Query Type 1 (Shortest Distance Over Frequent Communication Paths).**
   Given two integer person ids $p_1$ and $p_2$, and another integer $x$, find the minimum number of hops between $p_1$ and $p_2$ in the graph induced by persons who:
   (a) have made more than $x$ comments in reply to each others’ comments (see comment_hasCreator_person and comment_replyOf_comment);
(b) know each other (see person_knows_person, which presents undirected friendships between persons; a friendship relationship between persons $x$ and $y$ is represented by pairs $x\mid y$ and $y\mid x$).

2. **Query Type 2 (Interests with Large Communities).**
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:
(a) have that interest (see tag, person_hasInterest_tag);
(b) were born on $d$ or later;
(c) know each other (see person_knows_person, which presents undirected friendships between persons; a friendship relationship between persons $x$ and $y$ is represented by pairs $x\mid y$ and $y\mid x$).

3. **Query Type 3 (Socialization Suggestion).** 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 (see person_hasInterest_tag). For each of the $k$ pairs mentioned above, the two persons must be located in $p$ (see person_isLocatedIn_place, place, and place_isPartOf_place) or study or work at organizations in $p$ (see person_studyAt_organization, person_workAt_organization, organisation_isLocatedIn_place, place, and place_isPartOf_place). Furthermore, these two persons must be no more than $h$ hops away from each other in the graph induced by persons and person_knows_person.

4. **Query Type 4 (Most Central People).** 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:
(a) are members of forums that have tag name $t$ (see tag, forum_hasTag_tag, and forum_hasMember_person);
(b) know each other (see person_knows_person, which presents undirected friendships between persons; a friendship relationship between persons $x$ and $y$ is represented by pairs $x\mid y$ and $y\mid x$).

Here, the closeness centrality of a person $p$ is:
\[
\frac{(r(p) - 1) \cdot (r(p) - 1)}{(n - 1) \cdot s(p)},
\]
where $r(p)$ is the number of vertices reachable from $p$ (inclusive), $s(p)$ is the sum of geodesic distances to all other reachable persons from $p$, and $n$ is the number of vertices in the induced graph. When either multiplicand of the divisor is 0, the centrality is 0.

8 **Appendix: SNB Description**

Let’s briefly survey the SNB benchmark which was used during the contest and in the experimental section of this paper.

**The purpose.** In order to provide efficient evaluation for a variety of algorithms, tools, frameworks for social network data management tasks, a standard benchmark, called Social Network Benchmark (SNB) [16] was developed. This benchmark allows
not only efficient, but also a repeatable evaluation for a variety of scenarios: on-line transactions, business intelligence and graph analytics. Authors of the benchmark tried to make it as realistic as possible.

**Covered systems.** This benchmark covers several types of systems: graph DBMS and graph programming frameworks, RDF database systems, relational and NoSQL database systems.

**Data schema.** The general data schema of the benchmark is presented on Figure 1 (illustration taken from [16]). It is called Social Intelligence Benchmark Data Schema. The schema uses UML notation to describe entities, attributes and their relationships of different cardinalities. The schema defines the result of the benchmark’s data generator. Essentially it is a set of tables linked via primary-foreign key relationships.

The schema defines some social network and its most characteristic features:

1. users and their personal details, tags and likes;
2. relations between users (follows and knows);
3. textual content: posts and comment trees.

**Generator and its output: technical details.** This benchmark is essentially a synthetic data generator, which is implemented using MapReduce programming model. The generator is dictionary-based and is capable of generating correlated values. The result of the generator is the set CSV files, where each file contains records of the corresponding table.

**The benchmark and the contest.** The organizers of the contest used only the dataset generator, but not queries. Instead, they proposed four stand-alone types of queries, which we are describing in the corresponding section of the appendix.

The dataset generator provided four types of graph workloads: small (1k vertices), medium (10k vertices), large (100k vertices) and huge (1M vertices). The last one would be used for the final evaluation by the contest organizers.

Unfortunately, only the first two datasets were fully released to the public. The third one was discussed in the mailing list, where some of the generator parameters for this dataset were disclosed. However, no queries are known. In this paper we use the largest available (on the current date) dataset — the medium one for the experimental evaluation.

All of the queries are known at the start of the processing, contestants are not required to process them in a specific order.

9 Appendix: Experiments

Here we provide the graphs, illustrating the outcomes of our experimental evaluation. Note that Figure 2 and Figure 3 are put adjacent to each other and are situated on the same height. This allows us to examine the relative performance of these two approaches with respect to different number of threads.

10 Appendix: Miscellaneous Algorithms

This section describes miscellaneous Algorithms (Listing 8, Listing 10 and Listing 11), used for the evaluation of the query type 3.
Figure 1: Social Intelligence Benchmark Data Schema
Figure 2: Performance scalability (without pre-treatment phase).

Figure 3: Performance scalability and effects of query reordering (pre-treatment phase).

**Algorithm 9:** LOCATED\(\text{IN}\)

```plaintext
function located_in(places, p, sm)
    input: places — places of some person,
           p — given query parameter,
           sm — subplaces map
    output: true if places contain p or some of them is subplace of p

    ptype ← type of p;
    uniform ← ∅;
    if ptype == country then
        uniform ← sm[study places];
        uniform ← sm[location];
        uniform ← work places;
    else // ptype is city, therefore we don't need to look in work places that are countries
        uniform ← study places;
        uniform ← location;
    return p in uniform;
```
Algorithm 10: cBFS

```plaintext
define cBFS(source, cont, h):
    input: source — start vertex, cont — continent to search on, h — hops
    output: reached persons from cont
    /* Initialization */
    Enqueue(Q, source);
    for each friend of person do
        if hops[friend.id] == -1 then
            Enqueue(Q, friend);
            // set distance from source
            hops[friend.id] ← hops[person.id] + 1;
        if friend.continents ∋ cont then
            rp ← friend.id;
    while Q ≠ ∅ do
        person ← Dequeue(Q);
        if hops[person.id] == h then
            continue;
        foreach friend of person do
            if hops[friend.id] == − 1 then
                Enqueue(Q, friend);
                // set distance from source
                hops[friend.id] ← hops[person.id] + 1;
            if friend.continents ∋ cont then
                rp ← friend.id;
```

11 Appendix: Query Type 4 Evaluation Scheme

In this section we present the overall processing scheme for the query type 4 (see Figure 4).

![Figure 4: Query type 4 evaluation scheme](image-url)
Algorithm 11: gBFS

```plaintext
function gBFS(source, p, h)
  input: source — start vertex,
         p — place to search in,
         h — hops
  output: reached persons from p

  /* Initialization */
  Q ← source.id; // Queue
  hops ← array of -1 that size is size; // distances from source
  rp ← ∅; // reached persons

  Enqueue(Q, source);
  hops[source.id] ← 0;

  while Q ≠ ∅ do
    person ← Dequeue(Q);
    if hops[person.id] == h then
      continue;

    foreach friend of person do
      if hops[friend.id] == -1 then
        Enqueue(Q, friend);
        // set distance from source
        hops[friend.id] ← hops[person.id] + 1;

        if located_in(friend, p) then
          rp ← friend.id;
```

12 Appendix: Tables

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Persons} )</td>
<td>set of all people</td>
</tr>
<tr>
<td>( \text{Dates} )</td>
<td>set of all birthdays of people</td>
</tr>
<tr>
<td>( G_l \ldots h ), where ( l = \min \text{Dates}, h = \max \text{Dates} )</td>
<td></td>
</tr>
<tr>
<td>( N_f )</td>
<td>the average number of friends of person</td>
</tr>
</tbody>
</table>

Table 1: Summary of notation

<table>
<thead>
<tr>
<th>Year</th>
<th>1k count</th>
<th>10k count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>82</td>
<td>896</td>
</tr>
<tr>
<td>1981</td>
<td>102</td>
<td>991</td>
</tr>
<tr>
<td>1982</td>
<td>103</td>
<td>1009</td>
</tr>
<tr>
<td>1983</td>
<td>92</td>
<td>979</td>
</tr>
<tr>
<td>1984</td>
<td>115</td>
<td>994</td>
</tr>
<tr>
<td>1985</td>
<td>117</td>
<td>1000</td>
</tr>
<tr>
<td>1986</td>
<td>113</td>
<td>1012</td>
</tr>
<tr>
<td>1987</td>
<td>92</td>
<td>1045</td>
</tr>
<tr>
<td>1988</td>
<td>83</td>
<td>967</td>
</tr>
<tr>
<td>1989</td>
<td>93</td>
<td>1024</td>
</tr>
<tr>
<td>1990</td>
<td>8</td>
<td>103</td>
</tr>
</tbody>
</table>

Table 2: Distribution of people by year of birth

<table>
<thead>
<tr>
<th>File name</th>
<th>1k</th>
<th>10k</th>
</tr>
</thead>
<tbody>
<tr>
<td>comment_hasCreator_person</td>
<td>7 225</td>
<td>281 303</td>
</tr>
<tr>
<td>comment_replyOf_comment</td>
<td>6 233</td>
<td>239 184</td>
</tr>
<tr>
<td>forum_hasMember_person</td>
<td>8 459</td>
<td>251 887</td>
</tr>
<tr>
<td>forum_hasTag_tag</td>
<td>161</td>
<td>4 753</td>
</tr>
<tr>
<td>person</td>
<td>74</td>
<td>756</td>
</tr>
<tr>
<td>person_hasInterest_tag</td>
<td>30</td>
<td>334</td>
</tr>
<tr>
<td>person_isLocatedIn_place</td>
<td>8</td>
<td>92</td>
</tr>
<tr>
<td>person_knows_person</td>
<td>226</td>
<td>2 827</td>
</tr>
<tr>
<td>person_studyAt_organisation</td>
<td>10</td>
<td>117</td>
</tr>
<tr>
<td>person_workAt_organisation</td>
<td>29</td>
<td>322</td>
</tr>
<tr>
<td>place</td>
<td>60</td>
<td>191</td>
</tr>
<tr>
<td>place_isPartOf_place</td>
<td>8</td>
<td>26</td>
</tr>
<tr>
<td>tag</td>
<td>91</td>
<td>298</td>
</tr>
<tr>
<td>organisation_isLocatedIn_place</td>
<td>10</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 3: Files statistics