Distributed Join Processing on Social Network Data

Stefan Mengel
mengel@cril.fr

April 3, 2017

The goal of this project is to develop programs that process so-called join queries, one of the most important classes of database queries, in a distributed fashion. The programs are then evaluated on real-world data originating from social networks.

1 Introduction and some basic functionality

One recent challenge in graph algorithms that has emerged with the advent and of the internet and social networks is analysing the structure of huge graphs that arise from these networks. For example, graphs are used to describe links between web pages, friendship relations in social networks like google plus or facebook or collaborations between actors or scientists. One way to extract information from these graphs that has recently been proposed is counting the frequency of small subgraphs [5]. The enormous size of the input graphs suggests a distributed approach to this problem. Your task in this project is to develop a corresponding software and to evaluate it on real world data from social networks.

Finding small subgraphs in a graph is strongly related the problem of evaluating so-called join queries on databases and consequently we will use techniques developed for databases in this project. You will develop a program that allows to evaluate some simple join queries in a distributed fashion. This program will be written step by step in several tasks. Make sure to test you program with the given input data (see below) but also with handwritten inputs that you create yourself after the completion of every task to check the functionality you implemented.

Let us introduce some notions from the area of databases. A (finite) relation R is a set of tuples $(a_1, \ldots, a_r) \in D^r$ where D is a finite set called the domain. The value r is called the arity of the relation. Finite relations correspond directly to tables in databases. Of particular interest to us are binary relations, i.e., such in which all tuples contain exactly 2 elements, which correspond to (directed) graphs.

Task 1. We will work with data from real world networks that is taken from [4] but slightly adapted for convenience. The files can be found at https://www.enseignement.polytechnique.fr/profs/informatique/Stefan.Mengel/[1]. Each of the files contains an edge relation for a social network graph in the following format: The domain D consists of non-negative integers. Each line in the file gives one tuple in the relation where the entries are separated by one or more spaces. The arity of the relation is implicitly given by the number of entries in each line which we assume to be the same in all lines (and is actually 2 in all inputs).

Implement a datastructure that encodes relations of arbitrary arity and size. Make sure that this data structure allows to easily sort the entries and to iterate over the entries but also allows fast random access. Write functions that can read and write relations from and to files in the format described above. □

Task 2. Implement a function that orders a given relation lexicographically. This function takes as a second value a (suitably encoded) permutation of $\{1, \dots, r\}$ which gives the order in which the elements of the tuples are compared. For example, for the permutation $\left(\begin{smallmatrix}1&2\\2&1\end{smallmatrix}\right)$ we have that the tuple $(1, 2)$ is bigger than $(2, 1)$ while it is smaller for the permutation $\mathsf{id} = \left(\begin{smallmatrix}1&2\\1&2\end{smallmatrix}\right)$. □

2 Joins

Let $(z_1, \dots, z_r)$ be a tuple of (not necessarily different) variables and R a relation of arity r, then $R(z_1, \dots, z_r)$ is called an atom. We call a mapping a that assigns to the variables in $(z_1, \dots, z_r)$ values from D an assignment to $R(z_1, \dots, z_r)$. We say that a satisfies $R(z_1, \dots, z_r)$ if and only if $(a(z_1), \dots, a(z_r)) \in R$. We interpret an assignment as a tuple indexed by the variables in $(z_1, \dots, z_r)$, and consequently we see a set of assignments as a relation.

Let $R(z_1, \dots, z_r)$ and $R'(z'_1, \dots, z'_{r'})$ be atoms with relations R and R', respectively. The the join $R(z_1, \dots, z_r) \bowtie R'(z'_1, \dots, z'_{r'})$ is defined to be the set of assignments to the variables in $\{z_1, \dots, z_r, z'_1, \dots, z'_{r'}\}$ that satisfy both $R(z_1, \dots, z_r)$ and $R'(z'_1, \dots, z'_{r'})$. Intuitively, the join consists of the tuples that we get gluing together the tuples from R and R' that take the same value on their common variables.

As an example, let E be the edge relation of a graph G, then $E(x_1, x_2) \bowtie E(x_2, x_3)$ consists of the triples (v_1, v_2, v_3) of vertices from G such that v_1v_2 and v_2v_3 are edges in E.

Task 3. Implement a function that, given two relations R and R' and two corresponding lists of variables, computes the join. One somewhat efficient algorithms is as follows: Let X be the set of common variables and for a tuple t let $\pi_X(t)$ be the restriction of t onto X. Order the tuples of R and R' in such a way that their restrictions to the variables in X have the same order (use the function of Task 2 for this). Now iterate over the two relations in parallel starting at the first tuple in each and call the currently considered tuples t and t'. If t and t' coincide on X, add all

[1] Note that some of the datasets are quite large. If you have problems running your programs on these large instances in later stages of the project, feel free to consider subsets of the entry relations.

combinations of tuples that agree with t and t' on X (those are contiguous in your datastructure due to the order), add the result to the output and jump in R and R' to the first tuples that disagree with t and t' on X. If $\pi_X(t)$ and $\pi_X(t')$ are different two cases can occur: If $\pi_X(t)$ is lexicographically smaller than $\pi_X(t')$, go to the next tuple t. Otherwise, go to the next tuple t'.

Test your implementation! □

An expression of the form $R_1(\bar{x}_1) \bowtie R_2(\bar{x}_2) \bowtie \ldots \bowtie R_\ell(\bar{x}_\ell)$ is called a *multi-way join query* and it is evaluated by sequentially applying joins to two relations each. Note that the order in which we apply these joins does not matter for correctness since the join operation is commutative and associative. Still the choice of the order in which we evaluate may make a huge difference in the runtime of the computation.

As an example, the multi-way join query $E(x_1, x_2) \bowtie E(x_2, x_3) \bowtie E(x_1, x_3)$ computes all triangles in a graph.

We now have the means to compute all subgraphs of a certain type by join queries.

Task 4. Use your join function to compute $E(x_1, x_2) \bowtie E(x_2, x_3)$ and $E(x_1, x_2) \bowtie E(x_2, x_3) \bowtie E(x_1, x_3)$ for some handwritten inputs and some of the real world instances you were provided. Which result has more entries? Why? Up to which input size can you evaluate these join queries in reasonable time? □

3 Distributed joins

We will now evaluate join queries in a distributed fashion. The first approach that we follow is distributing the computation of the individual joins. To this end, we send all pairs of tuples that may potentially be joined to the same machine, then we apply the join routine from the last section on the machine and finally collect the result. If we want to compute more than a single join, we apply this method independently and sequentially for the individual joins.

We decide which tuples may potentially be joined by choosing one variable in the intersection of the two atoms to be joined and then deciding to which machine to send each tuple depending on the value of the chosen variable in the tuple. Clearly, if two tuples take different values in the chosen variable, they will never be joined, so we lose nothing by sending them to different machines. Also, every result of the join will be computed exactly once by this algorithm, so the collection process for the result is extremely simple.

Task 5. Implement the algorithm described above in MPI. For m machines, send a tuple with entry n at the variable you consider to the ($n \mod m$)th machine. Test and profile your implementation by computing all triangles in some of the given data files. How does it scale with the number of machines you use? □

One problem that one sometimes encounter when applying the algorithm above is the following: When not all possible entries appear in the tuples of a relation, then far more tuples may be sent to the one machine than to others which makes the computation unnecessarily slow. For example, say that $R = \{(0,1),(0,2),(2,3),(2,4),(4,0)\}$, we

have two machines and the first entry of the tuples decides to which machine they are sent. Then all tuples are sent to one machine while the other one stays idle.

One standard way of avoiding this problem are *hash functions* whose goal is roughly as follows[2]: Given n and m with $n \gg m$, a hash function is a function $f : \{1, \ldots, n\} \to \{1, \ldots, m\}$ such that every subset $S \subseteq \{1, \ldots, n\}$ gets roughly evenly distributed, i.e., for every $y \in \{1, \ldots, m\}$ we have $|f^{-1}(y) \cap S| \approx \frac{|S|}{m}$. In our setting, we use hash functions for the distribution of tuples to machines. Instead of sending a tuple with entry n to the machine $n \mod m$, we choose a hash function h and send the tuple to the machine $h(n)$.

Task 6. Find and use an implementation of hash functions and implement the join algorithm described above. Test it as before. Is there a runtime difference for the data you consider? □

Note that collecting the intermediate results after every join is actually not necessary. Instead, each machine can send its results directly to the machine where it will be needed in the next join.

Task 7. Implement the algorithm sketched above. Does it perform better than the previous algorithm? □

4 HyperCube join

One problem of the join algorithm as discussed in Section 3 is that when we evaluate a multi-way join query, the intermediate results may be far bigger than the final result. Since we have to distribute the intermediate results after each step, this may cause a high communication cost that slows down the computation.

The HyperCube join algorithm[1, 2] avoids this problem by not distributing the evaluation of the single joins but instead distributing the tuples to the machines in such a way that each machine evaluates the complete multi-way join query on its tuples and then only communicates back its end result. You will implement a simplyfied version of this algorithm.

Consider a query $R_1(\bar{z}_1) \bowtie R_2(\bar{z}_2) \bowtie \ldots R_\ell(\bar{z}_\ell)$. For simplicity, assume that all atoms are binary, so of the form $R_i(z_i, z_i')$. Let $X = \{x_1, \ldots, x_k\}$ denote the set of variables appearing in the atoms. Now let $m = m_1 \cdot m_2 \cdot \ldots \cdot m_k$ be the number of machines we want to use where the m_i are integers. Then we can identify each machine uniquely with an "address" in $[m_1] \times [m_2] \times \ldots [m_k]$ where $[m_i] := \{1, \ldots, m_i\}$. For example, consider the triangle query $E(x_1, x_2) \bowtie E(x_2, x_3) \bowtie E(x_1, x_3)$. Say we have $m = 27 = 3 \cdot 3 \cdot 3$ machines. Then the 27 machines are addressed by triples over the set $\{1, 2, 3\}$.

Now we choose for each variable x_i in X a hash function $f_i : [n] \to [m_i]$ independently from the others. Then we send each tuple $t = (t_1, t_2) \in R_i$ to the machines whose address coincides with $h_j(t_1)$ and $h_{j'}(t_2)$ on the variables x_j and $x_{j'}$ where x_j

[2]Note that there are different uses for hash-functions in particular in cryptography. Make sure to *not* use cryptographic hash-functions later as these have quite different aims.

and $x_{j'}$ are the variables of R_j. In our example, let $(3, 4)$ be a tuple in E. Then, for the atom $E(x_1, x_2)$, we send $(3, 4)$ to the machines $(h_1(3), h_2(4), 1)$, $(h_1(3), h_2(4), 2)$ and $(h_1(3), h_2(4), 3)$.

Now each machine evaluates the join query on the tuples it is sent. It is not hard that the results computed on the machines form a partition of the overall result which can then be collected.

Task 8. Implement the computation of all triangles in a graph with the algorithm described above. For simplicity, first choose $m_1 = m_2 = m_3$. Is this algorithm more efficient than that of the last section?

Now try different values for the m_i. How does this affect the runtime. Note that choosing the values m_i optimally is highly non-trivial [3]. Also try to evaluate bigger multi-way join query and compare to the program from the last section. Does the order in which you evaluate the joins influence the runtime? □

5 Some advice

In this section, you will find some advice for finishing this project successfully.

- As the ***-rating suggests, this is not an easy project. In particular, it will take you some time to develop, test, debug and evaluate your programs. Do *not* start working on this too late assuming that you can finish it in some overnight sprint sessions!
- Evaluating your programs is an important part of the project. Make sure that you run them on on inputs of significant and different sizes, vary the number of cores in the computer rooms you use (just using your own single computer is *not* enough) and other parameters if there are any. Document your findings and present them in the report, e.g. by giving figures with different curves.
- Your submission is supposed to contain runnable code. In particular, it should also contain explanations on how to compile and run your programs. If your programs cannot be compiled and run after reasonable effort, this will be considered the same as if the code was not submitted at all and graded accordingly.
- You are expected to show running programs during the soutenance. Prepare accordingly.
- If your program does not work as expected try to analyze why this is the case and fix it. If you do not find a solution (or only one that would take too much time to actually perform), discuss this in the report and during the soutenance. Showing that you are aware of problems and what you understand about them, will count positively.
- If you have problems during the project, feel to contact mengel@cril.fr. Just understand that replies might take some time,so if you ask something directly before the deadline, the answer might come too late.

References

[1] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In Richard Hull and Wenfei Fan, editors, *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 273–284. ACM, 2013.

[2] Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 212–223. ACM, 2014.

[3] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 63–78. ACM, 2015.

[4] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[5] Ömer Nebil Yaveroğlu, Noël Malod-Dognin, Darren Davis, Zoran Levnajic, Vuk Janjic, Rasa Karapandza, Aleksandar Stojmirovic, and Nataša Pržulj. Revealing the hidden language of complex networks. *Scientific reports*, 4, 2014.