CS 784: Foundations of Data Management **Spring 2021**

Lecture 3: The Computational Complexity of Relational Queries

Instructor: Paris Koutris

When we study a query language $\mathcal{L}$ from a theoretical perspective, we typically focus on the following two questions:

1. How expressive is $\mathcal{L}$, i.e., what properties can we express using the language?
2. What is the complexity of evaluating a query from $\mathcal{L}$?

In this lecture we study the complexity of evaluating queries, and in particular Conjunctive Queries, and Relational Algebra queries. Before that, we will start with a brief revision of a few basic computational complexity concepts.

3.1 A Short Introduction to Computational Complexity

When we refer to the computational complexity of a problem, we typically talk about a *decision problem*: given an input $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ of size n in bits, we have to decide whether the output is yes (output 1) or no (output 0). It will be helpful to define some complexity classes in terms of circuit complexity, which is why we next introduce the notion of a boolean circuit.

Definition 3.1 (Boolean Circuit)**.** *A* boolean circuit *C with n inputs is a directed acyclic graph. It contains n nodes with no incoming edges, which are called the input nodes, and one node with no outgoing edges, which is called the output node. All other nodes in the graph are called* gates*, and can be one of the following logical operations:* `OR`*,* `AND`*, and* `NOT`*.*

The fan-in *of a gate is the number of incoming edges. A* `NOT` *gate has always fan-in one, while the* `OR` *and* `AND` *gates can have fan-in at least* 2*.*

A boolean circuit computes an output from an input by pushing the input bits from the bottom of the circuit to the top (following the topological order of the nodes). We are interested in three parameters of a boolean circuit: (i) the *size*, which is the number of nodes in it, (ii) the *depth*, which is the longest directed path from an input node to the output node, and (iii) the *fan-in*, which is the maximum fan-in of an `OR` or `AND` gate in the circuit.

Exercise 3.2. *Construct a boolean circuit of depth 2 and unbounded fan-in that, given a string of n bits as input, decides whether it contains at least two 1's or not.*

We can now define our first class of interest:

Definition 3.3 (NC)**.** NC^i *is the class of problems solved using a boolean circuit of depth* $O(\log^i n)$*, and a polynomial number of gates of fan-in at most* 2*.*

We simply now define $\mathsf{NC} = \bigcup_{i \geq 0} \mathsf{NC}^i$. NC stands for Nick's Class. The class NC^0 is not very interesting, since it captures only functions that depend on a constant number of input bits. The following problems are in the class NC: integer multiplication, integer addition, and matrix multiplication. Another problem of interest is PARITY: *given an input string of size n, output yes if the string contains an even number of 1's, otherwise output no.*

Proposition 3.4. PARITY *is in* NC^1*.*

Similarly to NC we can define the class AC.

Definition 3.5 (AC)**.** AC^i *is the class of problems solved using a boolean circuit of depth* $O(\log^i n)$*, and a polynomial number of gates of unbounded fan-in.*

We also define $\mathsf{AC} = \bigcup_{i \geq 0} \mathsf{AC}^i$. One can show that $\mathsf{NC}^i \subseteq \mathsf{AC}^i \subseteq \mathsf{NC}^{i+1}$ (proving the statement is left as an exercise!). So we can write:

$$\mathsf{NC}^0 \subseteq \mathsf{AC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2 \ldots$$

It thus holds that NC = AC. The smallest class of AC is the class AC^0, which consists of circuits of constant depth, unbounded fan-in and polynomially many gates (in contrast to NC^0, the class AC^0 is contains many interesting problems!).

Why are we interested in the classes $\mathsf{NC}^i, \mathsf{AC}^i$? I turns out that we can equivalently describe NC^i as the class of decision problems solvable in time $O(\log^i n)$ on a parallel computer with a polynomial number of processors. The idea is that the gates at each level of the circuit can evaluate their outputs in parallel, and thus the running time will be equal to the depth of the circuit. Hence, NC can be thought as the class of problems that can be solved efficiently on a parallel computer. It is easy to see than that AC^0 is the "easiest" complexity class that we can parallelize, since we need only a circuit of constant depth! In other words, AC^0 is a very weak complexity class, which means that its expressive power is not strong. For example, the problem PARITY can not be expressed in this class.

Theorem 3.6. PARITY *is not expressible in* AC^0*.*

Another problem that is not expressible in AC^0 is the graph reachability problem STCON: *given a directed graph, and two nodes* s, t*, is there a directed path from* s *to* t*?*

Definition 3.7 (Logarithmic Space)**.** *A problem is in* L *if it can be computed by a deterministic Turing machine using* $O(\log n)$ *space. A problem is in* NL *if it can be computed by a non-deterministic Turing machine using* $O(\log n)$ *space.*

It is easy to see that STCON is in NL. In particular, it is NL-complete. Here, we have to be very careful on how we define a reduction to prove completeness for L or NL, since the reduction must

be a logarithmic-space reduction (and not a polynomial time). Another NL-complete problem is 2-satisfiability (satisfiability for a SAT formula where each clause has two variables). A complete problem for L is the undirected connectivity problem USTCON: given an undirected graph, is there a path from node s to node t?

The relationship between the complexity classes we have defined so far is as follows:

$$\mathsf{AC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}^2$$

Definition 3.8 (PSPACE)**.** *A problem is in* PSPACE *if it can be computed by a Turing machine using a polynomial amount of space.*

Including deterministic polynomial time (P) and non-deterministic polynomial time (NP), the hierarchy of the aforementioned complexity classes is as follows:

$$\mathsf{AC}^0 \subset \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}.$$

We only know that the first containment is strict; all the other class containment questions are big open problems in the area of computational complexity.

3.2 Complexity Notions for Queries

We now turn our attention to how complexity is defined for evaluating queries in the context of databases. Recall that so far we studied only decision problems, where the answer is *yes* or *no*. In the case of a boolean query, the correspondence to a decision problem is straightforward. In the case of non-boolean queries, we will distinguish between two different problems: (a) computing the full query result, i.e. the one that computes all tuples in the output, and (b) computing the decision problem of whether a tuple t belongs in the query output.

Let us now consider a CQ q that we execute over an input database I, where we want to compute all of $q(I)$. We start with a naive algorithm that computes the result. Let $k = |\mathbf{vars}(q)|$ be the number of variables in q, ℓ be the number of atoms in q, and $|q|$ be the sum of the arities of the atoms in the body of q. Note that $k, \ell \leq |q|$. Since a valuation maps each variable to a constant in $|\mathbf{adom}(I)|$, there can be at most $|\mathbf{adom}(I)|^k$ possible mappings. Observe that the quantity $|\mathbf{adom}(q)|$ is connected polynomially to the input size: $|\mathbf{adom}(I)| \leq |q| \cdot |I|$.

For each mapping, we can check whether it is a valuation or not in time linear to the number of atoms in q, and output the corresponding output tuple (we can achieve this by creating a hash table for each relation, and then probe the hash table). Hence, the running time of this simple algorithm is

$$O(|I| + \ell \cdot |\mathbf{adom}(I)|^k) = O(|q|^{k+1} \cdot |I|^k)$$

Looking at the above expression, we can observe the following things. First, if we assume that the query q is fixed (and hence only the database I is the input), the running time is *polynomial* in

the size of the input. On the other hand, if we fix the database I and have the query as input, or we have both the query and the database as inputs, the running time becomes exponential in the input. Since in practice I and q have very different behaviors (I is typically very large, while q is very small), it makes sense to distinguish the complexity of the evaluating a query depending on what is considered as an input.

We distinguish between 3 different types of complexity of evaluating a query, so that we can separate the influence of the query and the database on the complexity:

Data Complexity : the query is fixed, and the complexity is expressed in terms of the size of the database. This is useful in the context of databases, since the query is typically much smaller in size than the database.

Query Complexity : the database remains fixed, and the cost is expressed in terms of the size of the query. This complexity is not commonly considered in a database context.

Combined Complexity : the complexity is measured in the size of both the query and the database.

Observe that the combined complexity will always be at least as high as both the query and the data complexity. In this course we will mainly focus on data and combined complexity.

3.3 The Complexity of Conjunctive Queries

We showed that the naive algorithm of evaluating a CQ implies that the data complexity is in P. In fact, we can show that the data complexity for CQs (and actually any relational query) is in the much weaker complexity class AC^0.

Theorem 3.9. *The data complexity of evaluating a boolean CQ is* AC^0.

Proof. (Sketch) We construct a boolean circuit of constant depth (2) that simulates the naive algorithm that checks whether every possible mapping is a valuation. We first encode our input as a multi-dimensional boolean matrix (tensor), where the dimension is equal to the arity of the relation. In the first level of the circuit we have `AND` gates, where each `AND` gate corresponds to a mapping of variables to constants and is connected with the corresponding inputs. All the `AND` gates are connected to a single `OR` gate in the second level of the circuit. □

In fact, the data complexity of every query in RA (so evaluating FO formulas in general) is AC^0 (prove this more general result as an exercise!). This means that the data complexity for evaluating any SQL query is AC^0, a complexity class that as we discussed is easy to parallelize. This is why it is commonly said that SQL is *embarrassingly parallel*.

We have already shown that the *combined complexity* of evaluating a general CQ is NP-complete. This follows directly from the homomorphism theorem. In the next lecture, we will consider CQs for which we can have faster evaluation.

References

[Alice] S. ABITEBOUL, R. HULL and V. VIANU, "Foundations of Databases."