Parameterized Complexity for the Database Theorist

Martin Grohe
Division of Informatics, University of Edinburgh, Edinburgh EH9 3JZ, Scotland, UK.
Email: grohe@dcs.ed.ac.uk

1. Introduction

Parameterized complexity theory provides a framework for a fine-grain complexity analysis of algorithmic problems that are intractable in general. In recent years, ideas from parameterized complexity theory have found their way into various areas of computer science, such as artificial intelligence [15], computational biology [1, 21], and, last but not least, database theory [16, 19].

This short paper is a gentle introduction to the theory, focusing on the results most relevant for database theory. Interested readers are referred to Downey and Fellow's monograph [6] to learn more about parameterized complexity theory.

The paper is organised as follows: In Section 2 we describe two simple fixed-parameter tractable algorithms in an informal way. Section 3 presents the formal framework of parameterized complexity theory. Section 4 is a brief survey of the parameterized complexity of database query evaluation.

Interlude: A Database Theorist's Nightmare

WASHINGTON, DC. A small coffee shop near the headquarters of the NSF. Monday morning, 10.30. *"... and John, we've got to do something about database theory. It's really getting out of hand. I mean, look at the latest SIGMOD Record: 'Parameterized Complexity for the Database Theorist'. What on Earth is that supposed to be?" — "You're absolutely right. It's ridiculous. Do they think they're FOCS or what?" — "So what shall we do." — "Well, I don't really know. But here's an idea: We can let our new guy worry about this." — "Excellent. I'll talk to him later."*

And so poor Peter Parker, just transferred from the history division to the computer science division due to the latest budget cuts for the humanities, got his first assignment. If only he knew something about database theory. But isn't it always a good starting point to form a panel? Yes, a panel of ten leading database theorists which would then recommend the inevitable budget cuts in database theory. Sounds like the perfect solution ...

But now Peter has a new problem: How to form the panel? *"Let's beat the database theorists with their own weapons"*, so he thinks. *"We have this huge database of scientific publications. I'm sure I can find my panel there."* His first idea is that there might be ten people such that each paper in database theory has at least one author among these ten. Certainly any such ten people would form an excellent panel. OK, that may be a bit too optimistic. But maybe there are ten people such that every other database theorist has written at least one joint paper with one of the ten. That would be good enough. If it does not work out either, there will certainly be ten people such that no two of them have written a joint paper.[†] Having such people in the panel would at least guarantee a certain breadth.

Database Principles Column. Column editor: Leonid Libkin, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3H5, Canada. E-mail: libkin@cs.toronto.edu.

[†] It's a nice little exercise to figure out why.

Even nightmares sometimes have happy endings, and at this point a smile goes over the face of our sleeping database theorist. *"He wants to solve* SET COVER, DOMINATING SET, *and* INDEPENDENT SET. *Oh well, let him try.*[‡] *He will never find his panel, and there will be no budget cuts. Isn't theory great?"* And with these happy thoughts he wakes up. But there remain nagging doubts. And when he finds some time later, he decides to look at the problems more carefully.

2. SET COVER and related problems

SET COVER is the following problem:

Input:	A set V, a family F of subsets of V, and a positive integer k.
Problem:	Find a subset X of V of size k such that each set in F contains an element of X. (Such an X is called a *cover* of the family F.).

We can model Peter Parker's first idea to form a panel by SET COVER as follows: We let V be the set of all database theory authors. The family F (of sets of authors) contains, for each database theory paper, the set of authors of this paper. Then we ask for a set X of ten authors such that each set in F contains a member of X.

DOMINATING SET is the following problem:

Input:	A Graph $G = (V, E)$ and a positive integer k.
Problem:	Find a subset X of V of size k such that each vertex of G either is in X or has a neighbour in X. (Such an X is called a *dominating set* of G.).

To model Peter Parker's second idea to form a panel by DOMINATING SET, we form a graph whose vertices are all database theory authors with an edge between two authors if they have written a joint paper on database theory. Let us call this graph the *collaboration graph* (of database theory). A dominating set in the collaboration graph is a set of authors such that each author either belongs to the set or has written a joint paper with someone in the set.

Finally, INDEPENDENT SET is the following problem:

Input:	A Graph $G = (V, E)$ and a positive integer k.
Problem:	Find a subset X of V of size k such that no two elements of X are neighbours in G. (Such an X is called an *independent set* of G.).

To model Peter Parker's third idea to form a panel by INDEPENDENT SET, we note that an independent set in the collaboration graph is a set of authors such that no two of them have written a joint paper.

It is well-known that all three problems are NP-complete. So what worries our database theorist? The problem is that we are not really looking at the general problems as defined above, but that we are specifically asking for a set cover, dominating set, or independent set of size $k = 10$. And for fixed k all three problems are easily seen to be solvable in polynomial time. A straightforward algorithm just searches through all subsets X of V of size 10 and tests, for each such X, if it is a cover for the family F or a dominating set or independent set of the graph G. If we let n be the size of V and m the size of F, such an algorithm would have a running time of $O(n^{10} \cdot m)$ for finding a set cover of size 10, $O(n^{11})$ for finding a dominating set of size 10, and $O(n^{10})$ for finding an independent set of size 10. But even though these are polynomial time algorithms, they cannot be considered practical if n is large. In particular, for our database theory panel n is the number of database theory authors, and we can assume that this number is sufficiently large. So the $\Omega(n^{10})$ algorithms are no threat to database theory. The real problem is that there is no obvious reason to believe that there are no better algorithms. The fact that the problems are NP-complete in general does not help us

[‡] We'll explain the three problems and their relation to the three ways of forming a panel in the next section. All three problems are NP-complete, making it quite unlikely that there are algorithms solving them efficiently — *at least not in their full generality*.

in our situation where k is a fixed small number. If we look at the NP-hardness proofs, we see that they require a k of the same order of magnitude as n.

And indeed, at least for SET COVER there is an algorithm that does much better in our situation: This algorithm processes the elements of F (the author sets of database theory papers) one by one. It maintains a tree T whose edges are labelled by elements of V (authors). The height of this tree is at most k, the maximum size of the desired cover. Associated with each node t of T is a subset $X(t) \subseteq V$ consisting of all elements of V that occur as the label of an edge on the path from the root of T to t. At each stage during the execution of the algorithm, some leaves of the tree are *alive*. The algorithm maintains the following invariant:

- For all leaves ℓ of T that are alive the set $X(\ell)$ is a cover of all elements of F that have been processed so far.
- For each cover X of the elements of F that have been processed so far there is a leaf ℓ of T such that $X(\ell) \subseteq X$.

The algorithm is displayed in Figure 1.

SETCOVER(V, F, k)

1. Initialise T to be a tree with just one node r
2. *alive* $\leftarrow \{r\}$
3. **for all** $f \in F$ **do**
4.   *new_alive* $\leftarrow \emptyset$
5.   **for all** $t \in$ *alive* **do**
6.   **if** $f \cap X(t) \neq \emptyset$ **then**
7.   *new_alive* $\leftarrow$ *new_alive* $\cup \{t\}$
8.   **else if** $|X(t)| < k$ **then**
9.   **for all** $v \in f$ **do**
10.   add a new vertex t' to T and an edge from t to t' labeled by v
11.   *new_alive* $\leftarrow$ *new_alive* $\cup \{t'\}$
12.   *alive* $\leftarrow$ *new_alive*
13.   **if** *alive* $= \emptyset$ **then**
14.   **reject**
15. **return** a set X of size k such that $X \supseteq X(t)$ for some $t \in$ *alive*[§]

Figure 1.

Example 1. Suppose our algorithm is given the instance

$$\begin{aligned} V &= \{a, b, c, d, e, f, g, h\}, \\ F &= \big\{\{a, b, c\}, \{b, c, d\}, \{c, e, f\}, \{d, f\}, \{d, h\}, \{e, f, g, h\}\big\}, \\ k &= 3. \end{aligned}$$

and suppose that it processes the elements of F (in the main loop in lines 3–14) in the order listed.

Figure 2 shows the evolution of the tree T. Leaves that are alive are depicted in white. The top tree is T after the first two sets $\{a, b, c\}$ and $\{b, c, d\}$ have been processed. The bottom left tree is T after the four sets $\{a, b, c\}$, $\{b, c, d\}$,

[§]Here we are assuming that V has at least k elements, if this is not the case our algorithm rejects immediately.

$\{c, e, f\}$, $\{d, f\}$ have been processed. The bottom right tree is the T after all elements of F have been processed. The covers of F of size 3 represented by this tree are

$$\big\{\{a, d, e\}, \{a, d, f\}, \{b, e, d\}, \{b, f, d\}, \{b, f, h\}, \{c, d, e\}, \{c, d, f\}, \{c, d, g\}, \{c, d, h\}, \{c, f, h\}\big\}$$

(Note that the cover $\{c, d, f\}$ occurs on two branches of the tree.)

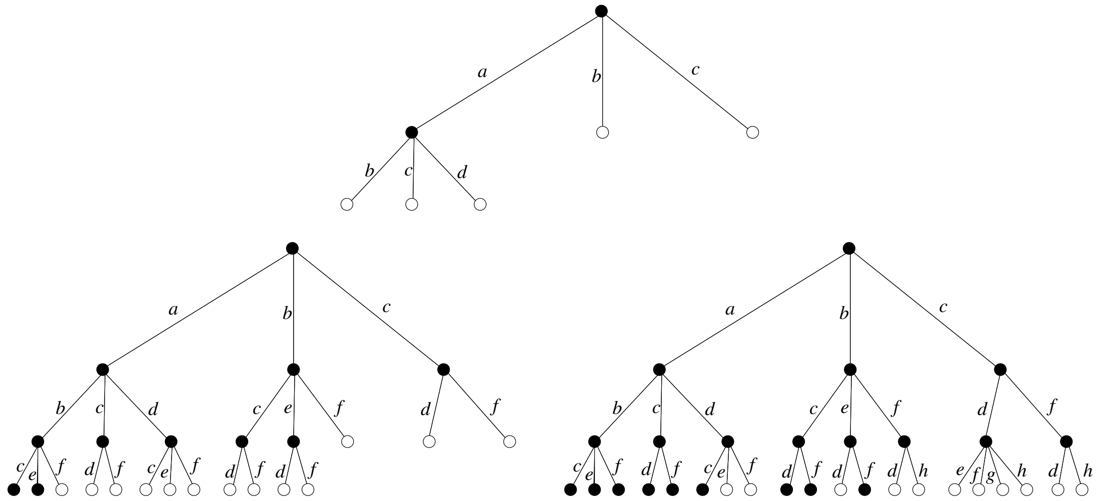

Figure 2. The evolution of the tree computed by the SET COVER algorithm of Figure 1 for the instance described in Example 1.

To analyse the running time of the algorithm, let $n = |V|$, $m = |F|$, and $s = \max\{|f| \mid f \in F\}$. Then each execution of lines 6–11 requires time $O(s)$. Observe that at any stage during the execution of the algorithm, T is an at most s-ary tree of height at most k. This means that during each execution the loop in lines 5–11 is iterated at most s^k times. Thus each execution of the loop requires time $O(s^{k+1})$. The outermost loop is iterated m times. Thus the overall running time of the algorithm is

$$O(s^{k+1} \cdot m).$$

In general, this running time is not better than the running time $O(n^k \cdot m)$ of the naive algorithm described before, because s can be as large as n. However, in many situations of practical interest, s is much smaller. In particular, for our database theory panel s is the maximum number of authors of a database theory paper. It is safe to assume that this number is not larger than 10.[¶] Then we have $s \leq 10$ and $k = 10$. Since n and m, the number of database theory authors and papers, respectively, are quite large, a running time of $10^{11} \cdot m$ is much better than one of $n^{10} \cdot m$. In particular, it is linear in the by far most significant parameters n and m.

Of course 10^{11} is still a very large constant. However, a simple heuristic optimisation will turn the algorithm into one that will work quite well in practice. Clearly, only very few database theory papers have 10 authors. Therefore, we have considerably overestimated the running time. Moreover, if we first sort the elements of the family F by size and then process them in this order, the chances are that our algorithm stops and rejects before it even has to look at the larger elements of F. In particular, it is quite likely that there are 11 singly authored database theory papers with different authors. In this case there cannot be a cover of size 10 and our algorithm will reject as soon as it has processed the first 11 singly authored papers by different authors.

So we have a reasonable algorithm for finding small set covers for (possibly very large) instances with a small set size s. What about dominating sets or independent sets? There is no obvious way to improve the naive $\Omega(n^k)$-algorithms. The assumption that the input graph is a collaboration graphs and the parameter s is small does not help

¶ If you know a database theory paper with more than 10 authors, well, then just make the assumption now for the sake of the argument.

at all, because it has no impact on the structure of the graph. However, a parameter that does help here is the *degree* d of the the input graph.[‖] Note that for our database theory panel application, d is the maximum of the number of co-authors of an author. This number can be quite large.[**] So a DOMINATING SET algorithm doing well on input graphs of small degree is less relevant for us.

Anyway, an algorithm very similar to the SET COVER algorithm described above solves DOMINATING SET in time

$$O((d+1)^k \cdot n).$$

We can use a different, but similarly simple, method to find small independent sets on graphs of small degree. Recall that the *distance* between two vertices of a graph is the length of the shortest path between these vertices. The *diameter* of a graph is the maximum of the distances between all pairs of vertices of the graph. Observe that every connected graph of diameter at least $2k-1$ has an independent set of size k. Thus to solve INDEPENDENT SET on connected graphs, we can proceed as follows: We are given a graph $G=(V,E)$ and a positive integer k. Let v_0 be an arbitrary vertex of G and V_0 the set of all vertices of distance at most $2k-1$ from v. Let G_0 be the induced subgraph of G with vertex set V_0. Then G has an independent set of size k if, and only if, G_0 has one. Thus it suffices to find an independent set of size k of G_0. Let $n_0=|V_0|$ be the number of vertices of G_0. We can use the naive algorithm to solve INDEPENDENT SET on G_0 in time $O(k^2 \cdot n_0^k)$. Observing that $n_0 \leq d^{2k}$, this gives us an

$$O(k^2 \cdot d^{2k^2} + n + m)$$

algorithm for INDEPENDENT SET on connected graphs. It can easily be extended to arbitrary graphs.

3. Fixed-parameter tractability and intractability

In the previous section we saw algorithms solving problems such as SET COVER or DOMINATING SET reasonably efficiently on possibly very large instances, provided that certain parameters of the input (k, s, d) are small.

We now give a formal framework for dealing with such *parameterized problems*. For simplicity, we only consider decision problems. $\mathbb{N}$ denotes the set of positive integers.

Definition 2. A *parameterized problem* is a set $P \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a finite alphabet.

If $(x,k) \in \Sigma^* \times \mathbb{N}$ is an instance of a parameterized problem, we refer to x as the *input* and to k as the *parameter*. The following two parameterized problems are two different *parameterizations* of the SET COVER problem:

Input:	A set V, a family F of subsets of V, and a positive integer k.
Parameter:	k.
Problem:	Decide if F has a cover of size k.

Input:	A set V, a family F of subsets of V, and a positive integer k.
Parameter:	$k+s$, where $s=\max\{|f| \mid f \in F\}$.
Problem:	Decide if F has a cover of size k.

The *standard parameterization* of an optimisation problem such as SET COVER, DOMINATING SET, or INDEPENDENT SET takes the size of the solution as the parameter. For example, the first of the two parameterizations of SET COVER is the standard parameterization, and the standard parameterization of DOMINATING SET is as follows:

Input:	A Graph $G=(V,E)$ and a positive integer k.
Parameter:	k.
Problem:	Decide if G has a dominating set of size k.

[‖] The *degree* of a graph is the maximum of the number of neighbours of all vertices.

[**] A famous example, although not from database theory, is Paul Erdös with 507 co-authors.

3.1. Fixed-parameter tractability.

Definition 3. A parameterized problem $P \subseteq \Sigma^* \times \mathbb{N}$ is *fixed-parameter tractable* if there is a computable function $f : \mathbb{N} \to \mathbb{N}$, a constant $c \in \mathbb{N}$, and an algorithm that, given a pair $(x, k) \in \Sigma^* \times \mathbb{N}$, decides if $(x, k) \in P$ in at most $f(k) \cdot |x|^c$ steps.

We denote the class of all fixed-parameter tractable problems by FPT.

The idea behind fixed-parameter tractability is that if we choose a parameterization of a problem in such a way that the parameter can be expected to be small in some situation we are interested in, then the dependence of the running time of an algorithm on the parameter is much less important than the dependence on the size of the input. Of course fixed-parameter tractability is only a rough approximation of the intuitive notion we are after. Clearly, an algorithm with a running time $O(2^{2^{2^{2^k}}} \cdot |x|)$ cannot be considered "tractable" in any real sense, not even for $k = 2$. However, similarly to polynomial time computability, fixed-parameter tractability is a robust and simple notion that overall captures a certain intuitive notion of tractability to a reasonable extent. We shall come back to this later (after Theorem 14).

We have seen in the previous section that the second parameterization of SET COVER (by $k+s$) is fixed-parameter tractable. Similarly, we have seen that the parameterizations of DOMINATING SET and INDEPENDENT SET by $k + d$ are fixed-parameter tractable. To show this, we used two simple algorithmic techniques that can be used to prove the fixed-parameter tractability of many other problems, too. For SET COVER and DOMINATING SET we used the fact that a set cover and dominating set can be found by exploring a *bounded search tree* whose size can be bounded solely in terms of the parameter. For INDEPENDENT SET we employed a method Downey and Fellows [6] call *kernelisation*: The original input instance is transferred to an instance of size bounded in terms of the parameter. On this smaller instance, called a *problem kernel*, the problem is solved by brute force.

Of course no parameterized complexity theory is needed to find the simple algorithms described in the previous section. The main purpose of the theory is to give evidence that certain problems are *not* fixed parameter tractable (just as the main purpose of the theory of NP-completeness is to give evidence that certain problems are not polynomial time computable).

3.2. Parameterized reductions.

Definition 4. Let $P \subseteq \Sigma^* \times \mathbb{N}$ and $P' \subseteq (\Sigma')^* \times \mathbb{N}$ be parameterized problems.

An *FPT-reduction* from P to P' is an algorithm that computes for every instance (x, k) of P an instance (x', k') of P' in time $f(k) \cdot |x|^c$ such that $k' \le g(k)$ and

$$(x, k) \in P \iff (x', k') \in P'$$

(for computable functions $f, g : \mathbb{N} \to \mathbb{N}$ and a constant $c \in \mathbb{N}$).

Example 5. *There is an FPT-reduction from the standard parameterization of* DOMINATING SET *to the standard parameterization of* SET COVER.

To see this, for a graph $G = (V, E)$ we let F_G be the family of all subsets $f_v = \{v\} \cup \{w \in V \mid (v, w) \in E\}$ for $v \in V$. The reduction maps the instance (G, k) of DOMINATING SET to the instance (V, F_G, k) of SET COVER.

Note that the same reduction is also a reduction from the parameterization of DOMINATING SET by $k + d$ to the parameterization of SET COVER by $k + s$.

The reduction given in the previous example is just the standard polynomial time reduction from DOMINATING SET to SET COVER. But be careful; not all polynomial time reductions are FPT-reductions. For example, INDEPENDENT SET can be reduced to SET COVER by mapping the instance $((V, E), k)$ of INDEPENDENT SET to the instance

$$\Big(V, \big\{\{v, w\} \mid (v, w) \in E\big\}, |V| - k\Big)$$

of SET COVER. This reduction is *not* an FPT-reduction from the standard parameterization of INDEPENDENT SET to the standard parameterization of SET COVER, because $|V| - k$ is not bounded in terms of k.

3.3. Parameterized intractability. Some combinatorial problems are provably not fixed-parameter tractable, and others, such as GRAPH COLOURABILITY parameterized by the number of colours, are not fixed-parameter tractable unless P = NP (because 3-COLOURABILITY is already NP-complete).

However, many interesting problems, such as the standard parameterizations of SET COVER, DOMINATING SET, and INDEPENDENT SET, do not seem to be fixed-parameter tractable, although there is no known way to prove this or reduce it to classical complexity theoretic questions such as $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$. To classify such problems, Downey and Fellows (cf. [6]) introduced a multitude of complexity classes above FPT. The most important of these are the classes of the so-called W-hierarchy

$$\mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \cdots .$$

The classes of the W-hierarchy are defined in terms of a parameterized version of the satisfiability problem for certain classes of formulas of propositional logic. We give this definition in detail below.

There is another parameterized complexity class called $\mathrm{AW}[*]$ that is important for us. It is defined, similarly as the classes of the W-hierarchy, in terms of a parameterization of the satisfiability problem for quantified propositional formulas. We omit the definition here and refer the reader to [6] or [16]. $\mathrm{AW}[*]$ contains all classes of the W-hierarchy. Intuitively, we may view the W-hierarchy as a ramification of a parameterized analogue of the classical complexity class NP and $\mathrm{AW}[*]$ as a parameterized analogue of the classical complexity class PSPACE.

3.4. The W-hierarchy. Formulas of propositional logic are build up from *propositional variables* $X_1, X_2, \ldots$ by taking conjunctions, disjunctions, and negations. The negation of a formula θ is denoted by $\neg\theta$. We distinguish between *small conjunctions*, denoted by $\wedge$, which are just conjunctions of two formulas, and *big conjunctions*, denoted by $\bigwedge$, which are conjunctions of arbitrary finite sets of formulas. Analogously, we distinguish between *small disjunctions*, denoted by $\vee$, and *big disjunctions*, denoted by $\bigvee$.

The *depth* of a propositional formula is the maximum number of nested conjunctions and disjunctions in the formula.

A formula is *small* if it only contains small conjunctions and small disjunctions. We define $\Gamma_0 = \Delta_0$ to be the class of all small formulas. For $t \geq 1$, we define Γ_t to be the class of all big conjunctions of formulas in Δ_{t-1}, and we define Δ_t to be the class of all big disjunctions of formulas in Γ_{t-1}. Note that the definitions of Γ_t and Δ_t are purely syntactical; every formula formula in Γ_t or Δ_t is equivalent to a formula in Γ_0. But the translation from a formula in Γ_t to an equivalent formula in Γ_0 usually increases the depth of a formula. For all $t, d \geq 0$ we let $\Gamma_{t,d}$ denote the class of all formulas in Γ_t whose small subformulas have depth at most d (equivalently, the whole formulas have depth at most $d + t$). We define $\Delta_{t,d}$ analogously.

The *weight* of a truth-value assignment S to the variables of a propositional formula is the number of variables set to TRUE by S. For any class Θ of propositional formulas, let the *weighted satisfiability problem for* Θ be the following parameterized problem:

Input:	$\theta \in \Theta$, positive integer k.
Parameter:	k.
Problem:	Decide if θ has a satisfying assignment of weight k.

Definition 6. For every $t \geq 1$, $W[t]$ is the class of all parameterized problems that are FPT-reducible to the weighted satisfiability problem for $\Gamma_{t,d}$ for some $d \geq 1$.

It is an immediate consequence of the definition of FPT-reductions that $\mathrm{FPT} \subseteq \mathrm{W}[1]$. Actually, it is conjectured that this inclusion is strict and that $\mathrm{W}[t]$ is strictly contained in $\mathrm{W}[t+1]$ for every $t \geq 1$.

Example 7. *The standard parameterisation of* INDEPENDENT SET *is contained in* W[1].

To see this, for every graph G we describe a propositional formula $\theta := \theta(G) \in \Gamma_{1,1}$ such that G has an independent set of size k if, and only if, θ has a satisfying assignment of weight k. It will be obvious from the construction that θ can be computed from G in polynomial time.

So let $G = (V, E)$ be a graph. For all $v \in V$, let X_v be a propositional variable. Let

$$\theta := \bigwedge_{(v,w) \in E} (\neg X_v \vee \neg X_w)$$

Then every satisfying assignment of θ corresponds to an independent set of G.

Theorem 8 (Downey and Fellows [4, 5]). *(1) The standard parameterization of* INDEPENDENT SET *is* W[1]*-complete under FPT-reductions.*

(2) The standard parameterizations of SET COVER *and* DOMINATING SET *are* W[2]*-complete under FPT-reductions.*

4. The parameterized complexity of database query evaluation

In this last section, we shall study the parameterized complexity of database query evaluation. For simplicity, we restrict our attention to Boolean queries (that is, queries with 'yes'/'no'-answer). However, all results extend to general queries, although sometimes these extensions require considerable efforts (see [9, 11] for details). For a query language L and a class C of database instances, the *query evaluation problem for* L *on* C is the following problem:

Input:	A (Boolean) query $\varphi \in$ L and a database instance $\mathcal{I} \in C$.
Problem:	Evaluate φ in $\mathcal{I}$ (i.e., decide if the answer is 'yes' or 'no').

For example, L may be SQL or the relational calculus and C the class of all relational databases. If we do not mention the class C of instances explicitly in the following, then C will always be the class of all relational databases. But we may also look beyond relational databases; for example, by letting L be XPATH and C the class of all XML-documents.

Traditionally, there are two important ways of measuring the complexity of query evaluation, *combined complexity* and *data complexity* (both going back to Vardi [24]). Combined complexity measures the complexity of the query evaluation problem simply in terms of the input size, that is, the size of the query plus the size of the database. It turns out that the combined complexity tends to be very high even for simple query languages. For example, the combined complexity of conjunctive queries is NP-complete [2] and the combined complexity of the relational calculus is PSPACE-complete [22, 24]. However, this high complexity is mainly caused by large queries; the size of the database instances is far less significant.* This is quite contrary to the situation we find in practice: We usually have to evaluate small queries against large databases. *Data complexity* measures the complexity of the query evaluation problem solely in terms of the size of the database and ignores the query size. Formally, the data complexity of a query language L is in a complexity class K if for all queries $\varphi \in$ L the problem of evaluating φ in a given instance $\mathcal{I}$ is in K. Naturally, data complexity tends to underestimate the "real" complexity of query evaluation. For example, a relational calculus query of size k can be evaluated in a database instance of size n in time $O(n^k)$. Thus the data complexity of the relational calculus is in PTIME. However, even if we argue that queries are small in practice and that therefore their size can be disregarded, a complexity of $O(n^k)$ can hardly be considered tractable even for small values of k (such as $k = 5$).

The discussion in the previous paragraph should ring a bell: Parameterized complexity theory offers a refined method for analysing situations exactly like this. We have to evaluate a query that can usually be expected to be small in a (potentially large) database. Therefore, the following parameterization of the query evaluation problem is appropriate:

Input:	A (Boolean) query $\varphi \in$ L and a database instance $\mathcal{I} \in C$.
Parameter:	Size of φ.
Problem:	Evaluate φ in $\mathcal{I}$.

Fixed-parameter tractability of the parameterized evaluation problem seems to be a more realistic notion of tractability than polynomial time data complexity, while at the same time it leaves a bit more flexibility than polynomial time combined complexity. The suggestion that parameterized complexity may be a suitable way of measuring the complexity of database query evaluation has first been made by Yannakakis [25].

The following theorem offers a nice alternative view on the fixed-parameter tractability of query evaluation. Intuitively, it says that fixed-parameter tractability of the query evaluation problem means that queries can be evaluated in polynomial time after being optimised first.

* Evaluating relational calculus queries is already PSPACE-complete on database instances of size 2.

Theorem 9 (Grohe [16]). *Let* L *be a query language and* C *a class of database instances. Then the query evaluation problem for* L *on* C *is fixed-parameter tractable if, and only if, there are algorithms* $\mathbf{O}$ *and* $\mathbf{E}$ *solving the problem as follows:*

(1) $\mathbf{O}$ *takes the input query* φ *and transforms it into some output* $\mathbf{O}(\varphi)$ *(intuitively, we can view* $\mathbf{O}$ *as a query-optimisation algorithm).*

(2) $\mathbf{E}$ *takes the input database instance* $\mathcal{I}$ *and the output* $\mathbf{O}(\varphi)$ *of the optimisation algorithm and computes* $\varphi(\mathcal{I})$*, the answer to the query, in time polynomial in the size of* $\mathcal{I}$*.*†

So is query evaluation fixed-parameter tractable for the standard query languages? In the relational setting, unfortunately the answer is 'no' (under certain complexity theoretic assumptions commonly believed to be true):

Theorem 10 (Downey, Fellows, Taylor [7]). *The parameterized query evaluation problem for the relational calculus is complete for the class* $\mathrm{AW}[*]$ *under FPT-reductions. Thus if* $\mathrm{FPT} \neq \mathrm{AW}[*]$ *then query evaluation for the relational calculus is not fixed-parameter tractable.*

Theorem 11 (Papadimitriou and Yannakakis [19]). *The parameterized query evaluation problem for conjunctive queries is complete for the class* $\mathrm{W}[1]$ *under FPT-reductions. Thus if* $\mathrm{FPT} \neq \mathrm{W}[1]$ *then query evaluation for conjunctive queries is not fixed-parameter tractable.*

There are a few positive results if we restrict the class of input database instances. All restrictions that have been considered refer to the graph structure of the instances. The *underlying graph*‡ of a relational database instance $\mathcal{I}$ is the graph whose vertices are all elements of the active domain of $\mathcal{I}$, with an edge between two vertices if they both occur in the same row of some table of $\mathcal{I}$.

Theorem 12 (Seese [20], Courcelle [3], Frick and Grohe [12], Flum and Grohe [10]). *Let* C *be a class of relational database instances such that the the graphs underlying the instances in* C *are either of bounded degree, or of bounded tree-width, or planar, or of bounded local tree-width, or have an excluded minor.*

Then the query evaluation problem for the relational calculus on C *is fixed-parameter tractable.*

For definitions of the graph properties mentioned in the theorem we refer the reader to [12].

Let us now look beyond the relational setting. It is well-known that XML-documents can be viewed as coloured trees (here colours represent XML-tags). Moreover, the core of the standard XML query languages is contained in monadic second order logic (see, for example, [18]). This makes the following well-known theorem sound quite interesting in the context of XML-query evaluation.

Theorem 13 (Folklore). *The query evaluation problem for monadic second-order logic on the class of coloured trees is fixed-parameter tractable.*

To prove this theorem, we can apply Theorem 9: The query-optimisation algorithm $\mathbf{O}$ transforms the given monadic second-order logic query into an equivalent tree-automaton. Such an automaton exists and can be effectively constructed by an old theorem due to Thatcher and Wright [23]. Then the evaluation algorithm $\mathbf{E}$ checks in time linear in the size of the input tree whether the automaton accepts the tree.§

Unfortunately, the algorithm described in the previous paragraph, while being linear in terms of the size of the input tree, is very inefficient in terms of the size of the input structure; it is actually *non-elementary*. Essentially, this means that is not bounded by the function

$$t_h(x) = \left.2^{2^{\cdot^{\cdot^{2^x}}}}\right\}\text{height } h$$

for any $h \geq 1$. Of course this does not rule out that there are other, better fixed-parameter tractable algorithms for evaluating monadic second-order queries on trees than the one using tree-automata. Unfortunately, this is not the case:

† We are assuming here that $\mathbf{E}$ has random access to the output $\mathbf{O}(\varphi)$ of $\mathbf{O}$. So $\mathbf{E}$ does not necessarily read the whole $\mathbf{O}(\varphi)$. Such an assumption is necessary if we want the running time of $\mathbf{E}$ to be bounded solely in terms of the size of $\mathcal{I}$.

‡ What we call underlying graph here is sometimes called the *Gaifman graph* or the *primal graph* of the instance.

§ There is a slight problem here in that XML-documents are unranked trees, whereas automata are usually considered on binary trees, but this can be handled easily by well-known techniques.

Theorem 14 (Frick and Grohe [13]). *Unless* $\mathrm{P} = \mathrm{NP}$*, there is no algorithm for evaluating monadic second-order queries on trees in time* $f(k) \cdot n^c$*, for any elementary function* f *and constant* c*. Here* k *denotes the size of the input query and* n *the size of the input tree.*

Similar lower bounds hold for the restricted evaluation problems for the relational calculus considered in Theorem 12. These lower bound results drastically show the limitations of a parameterized complexity analysis. Maybe admitting any computable function f as the parameter dependence in the definition of fixed-parameter tractability is too liberal after all, and some restriction, say, on singly exponential f, would be reasonable. Of course the negative results such as Theorem 10 and Theorem 11 would remain true for such a stricter notion of fixed-parameter tractability, but the positive results of Theorem 12 and Theorem 13 would not.

Let us close this paper with a truly positive fixed-parameter tractability result. Temporal logics such as LTL or CTL* are commonly used as specification languages in automated verification. They allow it to describe properties of paths in a structure (the state space of a system in verification). In this sense they are similar to languages such as XPATH which allow it to specify properties of paths in XML-documents. Indeed, Gottlob and Koch [14] have recently shown how the core of XPATH can be translated into a fragment of CTL*. The algorithms underlying the following theorem not only show that the evaluation problems for LTL and CTL* are fixed-parameter tractable, but they work efficiently in practice as well.

Theorem 15 (Lichtenstein and Pnueli [17], Emerson and Lei [8]). *The evaluation problems for* LTL *and* CTL* *on the class of all Kripke structures*[¶] *are solvable in time* $O(2^k \cdot n)$*, where* k *denotes the size of the input query and* n *the size of the input instance.*

References

[1] H.L. Bodlaender, R.G. Downey, M.R. Fellows, M.T. Hallett, and H.T. Wareham. Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences*, 11:49–57, 1995.

[2] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[3] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, pages 194–242. Elsevier Science Publishers, 1990.

[4] R.G. Downey and M.R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24:873–921, 1995.

[5] R.G. Downey and M.R. Fellows. Fixed-parameter tractability and completeness II: On completeness for $W[1]$. *Theoretical Computer Science*, 141:109–131, 1995.

[6] R.G. Downey and M.R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

[7] R.G. Downey, M.R. Fellows, and U. Taylor. The parameterized complexity of relational database queries and an improved characterization of $W[1]$. In D.S. Bridges, C. Calude, P. Gibbons, S. Reeves, and I.H. Witten, editors, *Combinatorics, Complexity, and Logic – Proceedings of DMTCS '96*, pages 194–213. Springer-Verlag, 1996.

[8] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[9] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. Currently available at `http://www.dcs.ed.ac.uk/~grohe`. A preliminary version of the paper appeared in *Proceedings of the 8th International Conference on Database Theory*, LNCS 1973, Springer-Verlag, 2001.

[10] J. Flum and M. Grohe. Fixed-parameter tractability, definability, and model checking. *SIAM Journal on Computing*, 31(1):113–145, 2001.

[¶] Kripke structures are coloured directed graphs; they are used in automated verification to model state spaces of the systems.

[11] M. Frick. Generalized model-checking over locally tree-decomposable classes. In H. Alt and A. Ferreira, editors, *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science*, volume 2285 of *Lecture Notes in Computer Science*, pages 632–644. Springer-Verlag, 2002.

[12] M. Frick and M. Grohe. Deciding first-order properties of locally tree-decomposable structures. *Journal of the ACM*, 48:1184 – 1206, 2001.

[13] M. Frick and M. Grohe. The complexity of first-order and monadic second-order logic revisited. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 215–224, 2002.

[14] G. Gottlob and C. Koch. Monadic queries over tree-structured data. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 189–202, 2002.

[15] G. Gottlob, N. Leone, and M. Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Logic Programming and Nonmonotonic Reasoning, 5th International Conference, LPNMR'99*, volume 1730 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1999.

[16] M. Grohe. The parameterized complexity of database queries. In *Proceedings of the 20th ACM Symposium on Principles of Database Systems*, pages 82–92, 2001.

[17] O. Lichtenstein and A. Pnueli. Finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on the Principles of Programming Languages*, pages 97–107, 1985.

[18] F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems*, pages 145–156, 2000.

[19] C.H. Papadimitriou and M. Yannakakis. On the complexity of database queries. *Journal of Computer and System Sciences*, 58:407–427, 1999.

[20] D. Seese. Linear time computable problems and first-order descriptions. *Mathematical Structures in Computer Science*, 6:505–526, 1996.

[21] U. Stege. *Resolving Conflicts in Problems from Computational Biology*. PhD thesis, ETH Zuerich, 2000. PhD Thesis No.13364.

[22] L.J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory*. PhD thesis, Department of Electrical Engineering, MIT, 1974.

[23] J.W. Thatcher and J.B. Wright. Generalised finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.

[24] M.Y. Vardi. The complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 137–146, 1982.

[25] M. Yannakakis. Perspectives on database theory. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, pages 224–246, 1995.