Contents

1

Database theory: Query languages

Nicole Schweikardt[1], Thomas Schwentick[2], Luc Segoufin[3]

[1]*Goethe-Universitat Frankfurt am Main;* [2]*Technische Universitat Dortmund;*
[3]*ENS de Cachan*

CONTENTS

Version of: April 6, 2011

Abstract   This chapter gives an introduction to the theoretical foundations of query languages for relational databases. It thus addresses a significant part of database theory. Special emphasis is put on the expressive power of query languages and the computational complexity of their associated evaluation and static analysis problems.

1.1   Introduction

Most personal or industrial data is simply stored in files and accessed via simple programs. This approach works well for small applications but is not generic and does not scale. New applications require new software, and classical software can hardly cope with huge data sets.

    Database management systems (DBMS) have been built to provide a generic solution to this issue. Usually, a DBMS enforces a clear distinction between how the data is stored on disk and how it is accessed via queries. When querying a database, one should not be concerned about how and where the data is physically stored, but only with the logical structure of the data. This concept is called the *data independence principle*.

Moreover a DBMS comes with an optimization engine containing evaluation heuristics, index structures, and data statistics that greatly improve the performance of the system and induce high scalability. Typical DBMS now handle gigabytes of data easily.

The roots of database theory lie in the work of Codd on the relational model, identifying the relational calculus and the relational algebra. Several other models were later proposed, e.g., the object oriented model and, more recently, the semi-structured model of XML. The relational model is now the most widely used by DBMS. For this reason, and for lack of space, we mainly consider the relational model in this chapter. Only in the last section we shall briefly discuss other major data models.

Theoretical database research has covered areas such as the design of query languages, schema design, query evaluation (clustering, indexing, optimization heuristics, etc.), storage and transaction management, and concurrency control to name just a few. In this chapter we only address the theory of query languages, which forms the core of database theory. For gaining an in-depth knowledge of database theory we refer the reader to the textbooks [3, 83, 2], to the survey articles that regularly appear in the *Database Principles Column* of *SIGMOD Record*, and to the Proceedings of the *ACM Symposium on Principles of Database Systems (PODS)* and the *International Conference on Database Theory (ICDT)*. For theoretical results on schema design and integrity constraints we further refer to Kanellakis' handbook chapter [69].

The intention of this article is not to give a complete historical account. Sometimes we favor more recent presentations over the original papers. Full references can usually be found in the cited work.

The present chapter is organized as follows: In Section 1.2 we introduce the relational model and its basic definitions. In the same section, we also describe the aspects of query languages we are mostly interested in: expressive power, evaluation complexity, and complexity of static analysis. In Section 1.3 we discuss the most basic relational query language, conjunctive queries. In Section 1.4 we describe the relational algebra and the relational calculus whose expressive power is usually considered a yardstick for relational query languages. We study restricted query languages that can be evaluated more efficiently and allow for automatic static analysis in Section 1.5. In contrast, in Section 1.6 we cover more expressive query languages which support recursion and counting. In Section 1.7 we conclude with a brief survey on query languages for some other data models.

1.2 General notions

In this section we define the general concepts used in database theory. We present the relational model and the notions of query and query language. We also introduce the key properties of query languages relevant for this article: their expressive power, how they are evaluated, their complexity, and optimization and static analysis that can be performed on them.

1.2.1 The relational model

In the early years of databases, when it became clear that file systems are not an adequate solution for storing and processing large amounts of interrelated data, several database models were proposed, including the hierarchical model and the network model (see, e.g., [102]).

One central idea at the time was that querying a database should not depend on how and where data is actually stored. This is known as the *data independence principle*.

One of the biggest breakthroughs in computer science came when Codd introduced the *relational model* in 1970 [25]. In this chapter we will focus on the relational model, as it is still dominating the databases industry. Furthermore, most classical results in database theory have been obtained for the relational model.

In a nutshell, the basic idea of relational databases is to store data in tables or, seen from a more mathematical point of view, in *relations*. Figure 1.1 displays our simple running example of a relational database with three relations containing information on operas. Each table header gives some structural information on the relation, called the *relation schema*. Formally, a relation schema R consists of the *relation name* (OPERAS in the first table), the number of columns, its *arity* arity(R), and names for each column, the *attributes*. The actual *content* of a relation is given by the set of rows of the table, the *tuples* of the relation. Each tuple has one entry for each attribute of the relation. We assume that each entry comes from a fixed, infinite *domain* of potential database elements. Elements in this domain are often called *constants* or *data values*.

A *database schema* σ is simply a finite set of relation schemas where no two relations have the same name. For the purpose of this article, we ignore the fact that a database schema usually also includes a set of integrity constraints like *key* or *foreign key* constraints. We refer to [69, 3] for a discussion of the theoretical issues raised by integrity constraints.

Finally, a *database instance* (or, for short, *database*) D over a database schema σ has one (finite) relation R^D of arity arity(R) for each relation schema R of σ. If the database instance is clear from the context we often write R instead of

Operas	Composer	Piece
	Puccini	Turandot
	Wagner	Tristan
	Britten	Peter Grimes
	Puccini	Tosca
	Monteverdi	Orfeo
	Bizet	Carmen

People	Artist	Type	Birthday
	Puccini	Composer	12/22
	Nilsson	Soprano	05/17
	Pavarotti	Tenor	10/12
	Bizet	Composer	10/25
	Callas	Soprano	12/2

Cast	Theatre	Piece	Artist
	Scala	Turandot	Pavarotti
	Bayreuth	Tristan	Nilsson
	Covent Garden	Peter Grimes	Vickers
	Met	Tosca	Callas
	Scala	Turandot	Raisa

FIGURE 1.1

An example database with three relations.

R^D. Each element in R^D is called a *tuple of R* in D. The set of values occurring in a database instance D, its *active domain*, is denoted by adom(D).

1.2.2   Queries

Of course, the main purpose of storing data in a database is to be able to query it. As an example, someone might be interested in knowing all the operas written by Puccini. The result of this query on our example database in Figure 1.1 consists of *Turandot* and *Tosca*. More precisely, in the terminology of relational databases, it consists of the relation with the two unary tuples having *Turandot* and *Tosca* in their *Piece*-column, respectively.

In general, a *query q* is just a mapping which takes a database instance D and maps it to a relation $q(D)$ of fixed arity. As we aim at evaluating queries by computers, we further require that this mapping be computable.

Some subtleties are associated with the term *computable query*, which we discuss next. First of all, the notion of computability is usually defined for functions mapping strings to strings. Thus, to fit this definition, we have to represent each database instance and each query result as a string. A complication arises from the fact that the tuples of a relation are unordered. —This is actually where the correspondence between tables and relations

fails: when we represent a relation by a table, we have to put the rows into some order. But this order is not considered as fixed for a particular relation. Thus, all row permutations of a table represent the same relation.— Coming back to the computability issue: there are at least as many strings representing a relation as there are permutations of its tuples. However, the result of the query should not depend on the particular order chosen in the encoding of a relation. More precisely, if one encoding is a permutation of the other, then the output for the one should be a corresponding permutation of the output for the other. Similarly, if the data values of a database are changed consistently, the values in the result should change accordingly. A query fulfilling all these requirements is called *generic*, and by *computable query* we actually mean *computable generic query* [*].

The following list of queries on our database example from Figure 1.1 will be used for illustrating the concepts introduced later.

(1) List the artists performing in an opera written by Puccini.

(2) List the theaters playing Puccini or Bizet.

(3) Is there an artist performing in at least two theaters?

(4) is any theater showing a piece whose composer's birthday is 12/22?

(5) Is any artist performing in an opera whose composer's birthday is the same as the artist's birthday?

(6) List all artists who never performed in an opera written by Bizet.

(7) List all artists who have performed in Bayreuth but neither at the Scala nor at the Met

(8) Is the total number of operas in the database even?

Apart from differences in the complexities of the queries, one can already observe a difference between queries with a yes/no answer, like queries (3) and (4) above, and queries that produce a set of tuples like queries (1) and (2). We refer to the former kind as *Boolean queries* [†].

1.2.3   Query languages

So far we have stated our example queries in natural language. This is (to date) not suitable for processing queries with a computer. Thus, there is a

---

[*] The first formal definition of a database query was given in [18]. A detailed treatment of the genericity issue and the question how constants can be handled is found in [3, 63].

[†] Technically, Boolean queries can be seen as 0-ary queries, where the answer "no" corresponds to the empty result set and the answer "yes" corresponds to the result set containing the empty tuple.

need for *query languages* that allow users to pose queries in a semantically unambiguous way. It is important to remark that one wants to avoid the use of general programming languages for querying databases for various reasons: they usually require more effort, they are error-prone, and they are not conducive to query optimization.

Ideally, a query language allows users to formulate their queries in a simple and intuitive way, without having any special proficiency in the technicalities of the database besides knowledge of the (relevant part of the) database schema. In particular, the user should not need to specify how the query is processed but only which properties the result should have. Query languages of this kind are called *declarative*.

The evaluation of a query is usually done in several stages:

(1) A *compilation* transforms it into an algebra expression (see 1.3.5),

(2) using heuristic rules, this expression is rewritten into one that promises a more efficient evaluation,

(3) from the latter expression, different query evaluation plans are constructed (e.g., taking into account different access paths for the data), and one of them is chosen based on statistical information on the actual content of the current database,

(4) this evaluation plan is executed using efficient algorithms for each single operation.

Several important issues arise:

Expressive power: What can and what cannot be expressed in the query language at hand?

Complexity of evaluation: How complex is it to actually evaluate the queries expressible in the query language?

Complexity of static analysis: How difficult is it to analyze and optimize queries to ensure a good evaluation performance?

Of course, SQL is the lingua franca for relational databases. Nevertheless, in this article we will concentrate on languages (e.g., the relational algebra and the relational calculus) which are better suited for theoretical investigations of the above-mentioned questions. Actually, these languages were developed first and SQL can be conceived as a practical syntax for the relational calculus. The compilation of SQL into the algebra is based on the fundamental result that the calculus can be translated into the algebra.

1.2.4 Expressive power

The *expressive power* of a query language is the set of queries it can express. This is an important measure for comparing different query languages. For instance, it can tell whether some features are redundant or not. Understanding the expressive power of a query language is a challenging task. Showing that a query is not expressible amounts to proving a lower bound, and lower bounds are often difficult to get.

Nevertheless, the close relationship of relational query languages with mathematical logic allows to apply methods from that field to gain insight in the expressive abilities and limitations of query languages. Indeed, there has been a strong mutual interaction with Finite Model Theory [81].

1.2.5 Evaluation and its complexity

It is not surprising that there is a trade-off between the expressive power of a query language and the computational resources needed to evaluate a query stated in this language. This *evaluation complexity* can be studied from different angles, depending on the scenario at hand. We will quickly describe some of these different aspects next.

The first distinction is between Boolean (yes/no) queries and queries with a relation as output. In the latter case one might ask any of the following two questions:

(a) What effort is needed to compute the full query result?

(b) Given a tuple, what effort is needed to determine whether it is contained in the query result?

Many of the complexity investigations concentrate on decision problems, thus they mostly deal with Boolean queries or with question (b) above. Nevertheless, many results can be easily transferred from Boolean queries to general queries. In fact, most query languages $\mathcal{Q}$ have the property that an algorithm for efficiently evaluating *Boolean $\mathcal{Q}$*-queries can be used to construct an algorithm that efficiently evaluates *arbitrary* (non-Boolean) $\mathcal{Q}$-queries: Given a database D and a query q whose result is a relation of arity r, a naive approach is to successively consider each possible result tuple $\bar{t}$, evaluate the Boolean query *"Does $\bar{t}$ belong to $q(D)$?"*, and output $\bar{t}$ if the answer is "yes". Then, however, the delay between outputting two consecutive tuples in $q(D)$ might be rather long, as a large number of candidate tuples (and according Boolean queries) might have to be processed prior to finding the next tuple that belongs to the query result. This delay can be avoided if, prior to checking whether tuple $\bar{t} = (t_1, \ldots, t_r)$ belongs to $q(D)$, the algorithm first checks whether $q(D)$ contains any tuple whose first component is t_1 — and if the answer is "no", tuples with first component t_1 will not be further considered. This way of exploring the space of potential result tuples leads to an

algorithm for computing $q(D)$ such that the delay between outputting any two consecutive tuples in $q(D)$ requires to process only a number of Boolean queries that is polynomial in the size of the database and the query q.[‡] In this way, an efficient algorithm for evaluating *Boolean* *Q*-queries leads to an efficient algorithm for evaluating *arbitrary* *Q*-queries (provided that *Q* satisfies some mild closure properties ensuring, e.g., that if $q \in Q$ and t_1 is a data value, then the query "Does $q(D)$ contain a tuple whose first component is t_1?" is also expressible in *Q*). Bearing this in mind, *Boolean* queries will be in the focus of our exposition.

We refer to the algorithmic problem of evaluating a Boolean query in a query language *Q* by Eval(*Q*).

The second distinction has to do with the durability of queries and databases. Sometimes the same query is posed millions of times against an ever changing database. It may therefore be "compiled" once and forever, and it is reasonable to spend quite some effort on optimizing the query evaluation plan. In this scenario, it makes sense to consider the query as a fixed entity and to express the complexity in terms of the size of the database only. This is called the *data complexity* of a query. A further interest in data complexity stems from the observation that very often the database is by magnitudes larger than the query.

In other scenarios, the database never changes, but lots of different queries are posed against it. Then, one is interested in measuring the cost in terms of the size of the query, this is called *query complexity*. In the most general scenario of *combined complexity*, the database changes and many different queries are asked, and therefore the complexity is measured in the size of both, the query and the database. For most query languages, the data complexity is considerably lower than the combined complexity, whereas the query complexity usually is the same as the combined complexity. We therefore will restrict attention to data complexity and combined complexity.

We express complexities in terms of standard complexity classes like PTime, NP, ExpTime, LogSpace, and PSpace. We will also mention some parallel complexity classes like AC^0 (the class of all problems solvable by uniform constant depth, polynomial size circuits with *not*, *and*, and *or* gates of unbounded fan-in), the class TC^0 (the analogue of AC^0 where also *threshold* gates are available), and LogCFL (the class of all problems that are logspace-reducible to a context-free language); for precise definitions we refer the reader to [109]. Recall that

$$AC^0 \subset TC^0 \subseteq \text{LogSpace} \subseteq \text{LogCFL} \subseteq \text{PTime} \subseteq \text{NP} \subseteq \text{PSpace} \subseteq \text{ExpTime}.$$

As mentioned before, it is usually a fair assumption that databases are big while the queries are small. Thus, algorithms which are bad in terms of

---

[‡]A systematic study of so-called *polynomial* (or even *constant*) *delay algorithms* has been initiated recently, see [51, 37, 9, 8, 29].

the size of the query but perform well in terms of the database size are often considered as feasible. A systematic way of studying phenomena of this kind is provided by the framework of *parameterized complexity*. We will present some results in this vein and refer to the (somewhat feasible) class FPT of *fixed parameter tractable* problems and the (presumably infeasible) classes W[1], W[P], and AW[∗], for which the following inclusions hold: FPT ⊆ W[1] ⊆ W[P] and W[1] ⊆ AW[∗]. For the precise parametric notions we refer to the survey [52] and the book [40].

1.2.6 Static analysis and its complexity

In the context of data complexity we already mentioned queries that are evaluated many times and therefore deserve to be optimized towards fast evaluation. Even queries that are evaluated once on very large data deserve optimizations. Query optimizers use cost models to decide what optimizations are worth doing. There are usually several ways of compiling a query into an evaluation plan and, even more, there are already several ways of expressing a query in the query language at hand. This often corresponds to several equivalent characterizations of the same query and sometimes induces radically different evaluation procedures after compilation. But which one should the system use? This task is attacked during query optimization.

Many optimization tasks rely on three simple questions: (1) Does query q ever produce a non-empty result? (2) Does query q_1 always produce the same result as query q_2? (3) Does query q_1 always produce a subset of the results of query q_2? We refer to the first question as the *satisfiability problem* (or, *non-emptiness problem*) to the second as the *equivalence problem*, and to the third as the *containment problem*. By Sat(Q) we denote the algorithmic problem of deciding for a given query in language Q whether there is at least one database on which the query has a non-empty result. We write Equiv(Q) and Cont(Q) for the problems of deciding whether for given queries q_1, q_2 from Q and every database D, $q_1(D) = q_2(D)$, respectively, $q_1(D) \subseteq q_2(D)$. We abbreviate the former by $q_1 \equiv q_2$ and the latter by $q_1 \subseteq q_2$.

Of course $q_1 \equiv q_2$ iff $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$. Furthermore, q_1 is satisfiable iff it is not equivalent to the query that always produces the empty result set. On the other hand, the equivalence problem reduces to the containment problem and, if the query language is closed under complementation and conjunction, the containment problem reduces to the emptiness problem.

The whole area of reasoning about semantic properties of queries is called *static analysis*. In terms of automatic static analysis, one is interested in finding out whether static analysis for a given query language is decidable at all and, if so, what its exact complexity is.

1.2.7   Conclusion

The expressive power, the complexity of evaluation, and static analysis are correlated properties of a query language. More expressive power usually increases the complexity of query evaluation and static analysis. But even if two query languages have the same expressive power, they may vastly differ in terms of the complexity of static analysis and query evaluation. The reason for this is that even if every query of one language can be translated into an equivalent query of the other language, the translation may turn a short query into a huge one. Thus, apart from expressive power and complexity, also the *succinctness* of queries is a natural measure for comparing query languages. Although well-investigated for languages used in specification and automated verification, a systematic study of the succinctness of database query languages has started only recently (see, e.g., [57, 33]). Somewhat related is Papadimitriou's work on the complexity of knowledge representation, where he compares the succinctness of various representation formalisms (see e.g. [87, 45]).

There is a trade-off between the expressive power of a query language and the complexity of query evaluation. As an example, if the data complexity of a query language is in PTime then this query language cannot express any NP-complete property unless PTime = NP. Succinctness considerations can help to investigate the relationship between different languages with respect to combined complexity.

In the end, designers usually try to get the best expressive power with the least complexity for the application area at hand.

1.3   The simplest language: conjunctive queries

We start with a simple query language, the conjunctive queries, whose expressive power is subsumed by most of the query languages we will consider and which is already able to express many common "every day queries". In particular, it corresponds to the very basic features of SQL.

After introducing the rule-based conjunctive queries, we study their expressive power and complexity. We then turn to a restriction, the *acyclic* conjunctive queries, for which query evaluation and static analysis have considerably lower complexity. Finally, we present a different mechanism, the *SPJR algebra*, for defining conjunctive queries.

1.3.1   Conjunctive queries: definition

We recall query (1) from Section 1.2.2, asking for all artists performing in an opera written by Puccini. In SQL this can be simply expressed by

        SELECT Artist
        FROM Cast, Operas
        WHERE Cast.Piece = Operas.Piece AND Composer = Puccini

This query can be expressed more concisely in the following rule-based way:

$$\text{PArtists}(x) \ :- \ \text{Cast}(z, y, x), \ \text{Operas}(\text{Puccini}, y) \tag{1.1}$$

This expression can be interpreted as a "tuple producing facility" in the following way: for each assignment of values to the variables x, y, z, for which (z, y, x) is a tuple in the Cast relation and $(\text{Puccini}, y)$ is a tuple in the Operas relation, (x) is a tuple in the result relation PArtists.

As an example, the assignment $x \mapsto$ Pavarotti, $y \mapsto$ Turandot, $z \mapsto$ Scala fulfills all requirements and produces the tuple (Pavarotti).

In general, a conjunctive query consists of a single *rule* of the form

$$Q(\bar{x}) :- R_1(\bar{t}_1), \cdots, R_\ell(\bar{t}_\ell)$$

where Q is the name of the relation that is defined by the query. Its *arity* is the arity of the tuple $\bar{x}$ (the PArtists query, for example, has arity one). The atom to the left of the symbol $:-$ is called the *head* of the query, whereas the expression to the right of the symbol $:-$ is called the *body* of the query. The tuples $\bar{x}$ and $\bar{t}_1, \ldots, \bar{t}_\ell$ consist of variables and/or constants. The *atoms* $R_i(\bar{t}_i)$ are such that R_i is the name of a relation occurring in the database schema and whose arity coincides with the length of the tuple $\bar{t}_i$. The $\bar{t}_i$ are not necessarily disjoint and altogether have to contain all variables of $\bar{x}$. The above query's *result* $Q(D)$ over a database D is obtained as follows: for each possible assignment α of values to the variables present in $\bar{x}, \bar{t}_1, \ldots, \bar{t}_\ell$ such that, for each $i \in \{1, \ldots, \ell\}$, the resulting tuple $\alpha(\bar{t}_i)$ belongs to the database relation R_i^D, the tuple $\alpha(\bar{x})$ is in $Q(D)$.

The set of all conjunctive queries is denoted by CQ. The reader should note that the name "conjunctive queries" is really justified: a variable assignment has to fulfill a conjunction of conditions to produce an output tuple.

1.3.2 Limitations of the expressive power of CQ

Given two databases D_1 and D_2 over the same schema, we write $D_1 \subseteq D_2$ if $R^{D_1} \subseteq R^{D_2}$, for each relation name R. A query q is said to be *monotone* if $D_1 \subseteq D_2$ implies $q(D_1) \subseteq q(D_2)$. For example, query (1) is monotone while query (6), asking for artists who never performed in an opera written by Bizet, is not: by adding the tuple (Scala, Carmen, Pavarotti) to the Cast relation, Pavarotti would no longer be in the result of that query.

It is not hard to obtain the following (see e.g. [3] for a proof):

THEOREM 1.1
CQ can define only monotone queries.

As a consequence, query (6) is not expressible in CQ. Even though monotonicity is a useful tool for testing non-expressiveness in CQ, it does not yield a complete characterization of CQ. In fact, there are monotone queries that cannot be expressed in CQ. An example is query (2) asking about theaters playing Puccini *or* Bizet.

1.3.3   Complexity of CQ

Next we discuss the complexity of evaluation and static analysis for conjunctive queries.

Evaluation.   The naive way of evaluating a conjunctive query is by trying out each possible variable assignment, resulting in about $|\text{adom}(D)|^k$ steps if k is the overall number of variables in the query. The complexity of this method is exponential in the number of variables and thus in the size of the query. If the query is considered fixed, however, it is polynomial in the size of the database. Using the terminology introduced in Section 1.2.5, the following complexity results hold:

THEOREM 1.2

(a) The data complexity of Eval(CQ) *is in* AC^0 *[65] and thus, in particular, in* LogSpace.

(b) The combined complexity of Eval(CQ) *is* NP-*complete [20].*

(c) The parameterized complexity of Eval(CQ) *(with the size of the query as parameter) is* W[1]-*complete [88].*

Statement (c) basically says that, as in the above naive algorithm, an exponent that grows with increasing k, can never be avoided. Specifically, it says that if the (widely believed) complexity theoretic conjecture "W[1] ≠ FPT" is true, then there is no algorithm that solves Eval(CQ) in time $f(k) \cdot |\text{adom}(D)|^c$, where f is an arbitrary computable function, k is the size of the input query, D is the input database, and c is an arbitrary integer. It is not difficult to see that statement (c) is equivalent to the following statement: if W[1] ≠ FPT, then there does not exist a pair (A_{opt}, A_{eval}) of algorithms such that A_{opt} is an algorithm of arbitrary complexity that optimizes an input conjunctive query q, and A_{eval} is an algorithm that takes as input the optimized query and a database D and computes the query's result $q(D)$ in time polynomial in the size of D (see [52] for details).

Statement (b) seems to indicate that the evaluation of conjunctive queries against a relational database is intractable, clearly contradicting everyday experience. Some explanations for this counter intuitive phenomenon will

be given in Section 1.4.3. The proof of statement (b) is by a straightforward reduction of the NP-complete clique problem for undirected graphs to the query evaluation problem for conjunctive queries: an input instance (G, k) for the clique problem is simply mapped to a database representing the graph G and a conjunctive query asking whether G contains a clique of size k.

Static Analysis. It is not difficult to see that every conjunctive query $q :=$ $Q(\bar{x}) : -R_1(\bar{t}_1), \cdots, R_\ell(\bar{t}_\ell)$ is satisfiable. In fact, there is a *canonical database D_q* witnessing this: to construct D_q just put, for every i, the tuple $\bar{t}_i$ into relation R_i. Then, $Q(D_q)$ at least contains the tuple $\bar{x}$. E.g., the canonical database for the example query (1.1) has the tuple (z, y, x) in Cast and (Puccini, x) in Operas, hereby viewing x, y, z as ordinary data values. Thus, the problem Sat(CQ) is trivially solvable, since *every* conjunctive query is satisfiable.

As a matter of fact, a very similar approach also works for Cont(CQ):

THEOREM 1.3 [20]
Let $q_1(\bar{x})$ and $q_2(\bar{x})$ be two conjunctive queries with the same free variables $\bar{x}$. Then $q_1 \subseteq q_2$ if and only if $\bar{x} \in q_2(D_{q_1})$.

This theorem is very often stated in terms of homomorphisms: Given two databases D_1 and D_2 we say that $h : \text{adom}(D_1) \to \text{adom}(D_2)$ is a *homomorphism* from D_1 to D_2 if, for each relation R, $\bar{a} \in R^{D_1}$ implies $h(\bar{a}) \in R^{D_2}$. The *homomorphism theorem* [20] then states that, with the notation of Theorem 1.3, $q_1 \subseteq q_2$ if and only if there is a homomorphism from D_{q_2} to D_{q_1} fixing $\bar{x}$. The following is a corollary of Theorem 1.3 and Theorem 1.2 (b).

THEOREM 1.4 [20]
The containment problem Cont(CQ) *and the equivalence problem* Equiv(CQ) *are* NP-*complete.*

Furthermore, conjunctive queries can be minimized in the following sense: There is an algorithm which takes as input a conjunctive query q and outputs an equivalent conjunctive query q' such that the number of atoms in the body of q' is as small as possible (cf., e.g., the textbooks [102, 3] for details).

1.3.4 Acyclic conjunctive queries

We have seen that the combined complexity and the parameterized complexity of evaluating conjunctive queries are NP-complete and W[1]-complete, respectively, and thus the worst case complexity can, to the best of our knowledge, be expected to be exponential in the size of the query. Nevertheless, for practical purposes and if the query structure is "simple", there are smarter

evaluation algorithms than the naive "test all possible variable assignments" approach mentioned above.

To illustrate this, let us consider the query (4) asking whether any theatre plays a piece whose composer's birthday is 12/22. We can express this by the Boolean conjunctive query

$$\text{BAnswer}() :- \text{Cast}(z, y, x), \text{Operas}(x', y), \text{People}(x', \text{Composer}, 12/22).$$

One possible evaluation plan for this query is as follows: First, combine Cast and Operas to obtain an intermediate relation R in the following way: combine each tuple in Cast with all tuples in Operas that have the same entry in the Piece column (due to the two occurrences of y). For our example database, the resulting relation R is shown in Figure 1.2. Afterward, let S be the relation (also shown in Table 1.2) that consists of all tuples $\bar{t}$ from R for which there is a tuple in People with entries Composer and 12/22 in the Type and Birthday column and whose entry in the Artist column coincides with $\bar{t}$'s entry in the Composer column. Finally, the answer returned by the BAnswer query is "yes" if and only if the relation S is non-empty.

R	Theatre	Piece	Artist	Composer
	Scala	Turandot	Pavarotti	Puccini
	Bayreuth	Tristan	Nilsson	Wagner
	Covent Garden	Peter Grimes	Vickers	Britten
	Met	Tosca	Callas	Puccini
	Scala	Turandot	Raisa	Puccini

S	Theatre	Piece	Artist	Composer
	Scala	Turandot	Pavarotti	Puccini
	Met	Tosca	Callas	Puccini
	Scala	Turandot	Raisa	Puccini

FIGURE 1.2

Intermediate results for the BAnswer query.

The evaluation order of this strategy can be depicted as a tree as shown in Figure 1.3 (a). Note that in this tree, the relation atoms of the query occur at the leaves, and the inner nodes correspond to intermediate results. From the root, the final result can be obtained by dropping some (maybe all or none) of the columns.

Even though it does not hurt in this small example, it could be annoying that the arity of the intermediate relation R is larger than the arities of the input relations. Furthermore, R is basically the cartesian product of two relations, joined in their Piece column. For a larger query this might result in a wide table which on a "real life" database could grow very large.

$$S(z,y,x,x')$$

$$R(z,y,x,x') \qquad \textsc{People}(x',\text{Composer},12/22)$$

$$\textsc{Cast}(z,y,x) \qquad \textsc{Operas}(x',y)$$

(a)

$$S'(x',y)$$

$$R'(x',y) \qquad \textsc{Cast}(z,y,x)$$

$$\textsc{Operas}(x',y) \qquad \textsc{People}(x',\text{Composer},12/22)$$

(b)
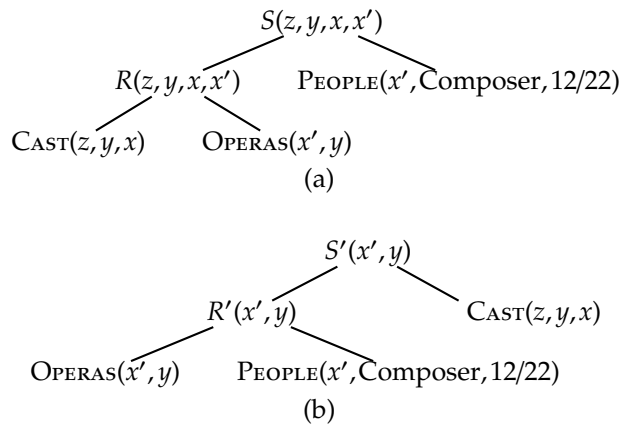
FIGURE 1.3

Two different evaluation trees for the PSoprano query.

Nevertheless, a closer look at the query shows that these problems can be avoided when using a different evaluation plan, depicted in Figure 1.3 (b). Even though at first sight it looks similar to the original plan, it has an important extra property: all tuples that enter the intermediate relation R' have to be tuples from relation Operas, and the same holds true for relation S'. In particular, the intermediate relations do not grow in arity, and their content is never obtained by an expensive product operation but rather by a kind of filter operation selecting tuples from one relation, controlled by the other relation. One could say that each intermediate relation is *guarded* by one of the input relations. Evaluation trees with this property are sometimes called *join trees*, and queries for which such a join tree exists, are called *acyclic*. We denote the class of acyclic conjunctive queries by ACQ. For example, query (4) is acyclic whereas query (5) is not.

If a join tree is given for a Boolean query q, the query can be evaluated by processing the join tree bottom-up. It is easy to see that each intermediate step can be performed in polynomial time. As it is possible to test in polynomial time (even in LogSpace [47, 91]) whether a join tree exists and to actually compute one if this is the case, it follows that Eval(ACQ) is in PTime [111].

Note that the intuition given above is accurate only for *Boolean* queries. But analogous notions of join trees and acyclic queries also exist for non-Boolean queries (see [3] for details) that can be evaluated efficiently by processing a join tree in a bottom-up phase followed by a top-down phase and, possibly, another bottom-up phase (see [111, 39] for an explanation of the bottom-up and top-down phases). The class of acyclic queries has been characterized in various ways (cf. [3]); the term "acyclic" is due to a characterization referring to acyclic hypergraphs [12].

A precise complexity analysis yields the following result, showing that

query evaluation and static analysis of acyclic queries not only belong to PTime but can even be efficiently parallelized:

THEOREM 1.5 [47]

(a) The combined complexity of Eval(ACQ) *is* LogCFL*-complete.*

(b) The containment problem Cont(ACQ) *is* LogCFL*-complete.*

In this sense acyclic queries behave nicely, and it is natural to wonder whether there are further classes of well-behaved queries. Indeed, there are several extensions and variations of the notion of acyclicity.

One line of variations is based on the *tree-width* of query graphs (see [47, 39] for the basic concept and references to the literature). For a conjunctive query q, the *graph of q* is the graph whose vertices are the variables of q and which has an (undirected) edge between two vertices whenever the corresponding variables occur in the same atom in the body of q. Given a class C of graphs, let CQ[C] be the set of conjunctive queries q such that the graph of q is in C. For instance, if $\mathcal{A}$ is the set of acyclic graphs, then CQ[$\mathcal{A}$] is a subset of ACQ. In order to generalize this we let, for each number k, CQ_k be the class of conjunctive queries whose graph has tree-width at most k. It is known that, for any number k, an analogue of Theorem 1.5 holds (a PTime upper bound was obtained already in [23]), and static analysis is also tractable for conjunctive queries of bounded tree-width:

THEOREM 1.6 [47]

(a) For any number $k \geq 1$, the combined complexity of Eval(CQ_k) *is* LogCFL*-complete.*

(b) For any number $k \geq 1$, Cont(CQ_k) *is* LogCFL*-complete.*

When the schema is fixed, relative to a plausible complexity theoretic assumption, even a *precise* characterization of the *tractable* graph-based classes of conjunctive queries is known:

THEOREM 1.7 [58]
Assume that W[1] $\neq$ FPT *and let C be a recursively enumerable class of graphs. Then the combined complexity of* Eval(CQ[C]) *is in* PTime *if and only if C has bounded tree-width.*

Even more, an extension of the above result providing a complete characterization of all tractable classes of conjunctive queries over a fixed schema has been obtained in [53].

When (the arity of) the schema is not fixed, a larger class of tractable queries can be found by considering the *hypergraph* of a query instead of the graph. The *hypergraph of query q* has one hyperedge per query atom *A* (in the body of *q*) which contains all nodes corresponding to variables in *A*. In [48], the concepts of *hypertree decompositions* and *hypertree-width* were introduced. It was shown that the acyclic conjunctive queries are precisely the queries whose hypergraph has a hypertree decomposition of hypertree-width 1, and that for each number $k \geq 1$, the statement of Theorem 1.5 (a) can be generalized from ACQ to the class of all conjunctive queries that have a hypertree decomposition of hypertree-width at most k. Various other classes of tractable queries based on hypergraph decompositions have been proposed in the literature; we refer to [94] for a survey.

1.3.5 Relational view

So far we considered only SQL and rule-based conjunctive queries as query languages. They are both declarative in the sense that they specify answers by their properties rather than by operations used to construct them. We already talked about evaluation plans in Section 1.3.4. A concise way to express evaluation plans is offered by the *relational algebra*. It consists of a few simple operators for manipulating relations. For the following it is useful to think of a relation as a table where each row (column) of the table corresponds to a tuple (attribute) of the relation.

The operators of the relational algebra that are needed for expressing conjunctive queries are (1) extraction of rows of a table (*selection*), (2) extraction of columns of a table (*projection*), (3) gluing together two tables along some columns (*join*) and (4) renaming of columns. For example, $\sigma_{\text{Composer='Puccini'}}(\text{OPERAS})$ is a selection which extracts all rows where the composer (i.e., the first attribute) is 'Puccini'. The expression $\pi_{\text{Theatre,Piece}}(\text{CAST})$ extracts the first two columns of the CAST relation. Note that the resulting table only has four rows. The expression CAST ⋈ OPERAS is a join which combines each row of the CAST table with each row of the OPERAS table, provided they have the same value, in each column with the same name, i.e., provided they match on the Piece attribute. Thus, the resulting table is the relation R depicted in Figure 1.2; it has four columns and five rows. Query (1) can thus be expressed by $\pi_{\text{Artist}}\big(\sigma_{\text{Composer='Puccini'}}(\text{CAST} \bowtie \text{OPERAS})\big)$, whereas the query asking for all sopranos who perform in an opera written by Puccini is expressed by

$$\pi_{\text{Artist}}\big(\sigma_{\text{Composer='Puccini'}}(\text{CAST} \bowtie \text{OPERAS}) \bowtie \sigma_{\text{Type='Soprano'}}(\text{PEOPLE})\big).$$

The algebra consisting of the four operators of selection, projection, join,

and renaming is called the *SPJR algebra*. Note that the SPJR algebra can express unsatisfiable queries, e.g., $\sigma_{\text{Composer}='\text{Puccini}'}(\sigma_{\text{Composer}='\text{Verdi}'}(\text{OPERAS}))$ will never return any opera. There is a straightforward polynomial time algorithm to test whether an SPJR query is satisfiable.

However, the ability to express unsatisfiable queries is the only difference between SPJR and CQ with respect to expressive power. It is not difficult to see the following (cf. [3] for details).

THEOREM 1.8
CQ and satisfiable SPJR have the same expressive power. Moreover, queries can be translated from either language to the other in polynomial time.

Recall that basic SQL queries are formed using the following syntax:

> select *attributes*
> from *relations*
> where *test*.

Conjunctive queries are essentially those which can be expressed by SQL queries where the *test* part contains only conjunctions of equality tests.

1.4 The gold standard: Codd-complete languages

We recall from Section 1.3.2 that conjunctive queries can only express monotone queries, i.e. queries which never produce smaller result sets if something is added to the database. Of course, there are interesting non-monotone queries, e.g. query (6) from Section 1.2.2. Furthermore, simple disjunctions like query (2) are not expressible in CQ either. In this section we consider extensions of CQ that are capable of expressing such queries: Codd-equivalent languages such as the relational algebra, the relational calculus, and the non-recursive rule-based queries. After introducing these query languages, we study their expressive power and their complexity.

1.4.1 Codd-equivalent query languages: definition

The rule-based queries of Section 1.3.1 can be extended by (1) allowing queries to consist of more than one rule, (2) allowing relations defined by one or more rules to be used in the body of other rules, and (3) allowing negated atoms in the body of rules. A query is thus a finite set of *rules* of the form

$$Q(\bar{x}) :- A_1(\bar{t}_1), \cdots, A_\ell(\bar{t}_\ell)$$

where the atoms $A_i(\bar{t}_i)$ are either of the form $S(\bar{t}_i)$ or of the form $\neg S(\bar{t}_i)$, where S is either a relation name of the database schema or a the name of a relation symbol used in the head of one of the rules of the query [§].

Nevertheless, recursion is not allowed. More formally, the following directed graph is not allowed to contain a directed cycle: the graph's nodes are the relation names, and there is an edge from R_1 to R_2 if R_1 appears in the body of a rule having R_2 in its head.

As an example, the following rules

$$
\begin{aligned}
\textsc{Stars}(x) \ &:- \ \textsc{Cast}(\text{Scala}, y, x) \\
\textsc{Stars}(x) \ &:- \ \textsc{Cast}(\text{Met}, y, x) \\
\textsc{Result}(x) \ &:- \ \textsc{Cast}(\text{Bayreuth}, y, x), \ \neg\textsc{Stars}(x)
\end{aligned}
$$

describe query (7) from Section 1.2.2, selecting all artists who have performed in Bayreuth but neither at the Scala nor at the Met. We call such queries *non-recursive rule-based queries*.

The same expressive power can be obtained in two other ways:

(1) By adding the relational *union* and *difference* operators to the SPJR algebra one gets the full *relational algebra*.

(2) The *relational calculus* consists of the logical formulas of the predicate calculus (i.e., *first-order logic* FO) which use the relations of the database schema plus equality as relation symbols and do not use any function symbols. We will sometimes briefly write FO to denote the relational calculus.

Query (7) from Section 1.2.2, for example, can be expressed by the relational algebra expression

$$\pi_{\text{Artist}}(\sigma_{\text{Theatre='Bayreuth'}}(\textsc{Cast})) - \pi_{\text{Artist}}(\sigma_{\text{Theatre='Met'}}(\textsc{Cast}) \cup \sigma_{\text{Theatre='Scala'}}(\textsc{Cast}))$$

and by the relational calculus formula $\varphi_{\textsc{Result}}(x) :=$

$$\Big(\exists y \, \textsc{Cast}(\text{Bayreuth}, y, x) \ \wedge \ \forall y \, \neg \big(\textsc{Cast}(\text{Scala}, y, x) \vee \textsc{Cast}(\text{Met}, y, x) \big) \Big)$$

When fixing the precise semantics of the relational calculus, some care needs to be taken to decide over which domain variables should range. One possible solution is to let the quantifiers range only over elements in the active domain of the underlying database; another way is to let them range over the entire domain of potential data values but to restrict the syntax in a "safe" way to avoid infinite query results and to ensure efficient evaluation. A similar problem occurs in the context of negations in non-recursive rule-based queries. Again, to avoid infinite query results, one can either restrict

---

[§]Recall from Section 1.3.1 that the *body* of a rule consists of the atoms to the right of the symbol $:-$ and the *head* of a rule is the atom to the left of the symbol $:-$.

attention to the active domain of the underlying database or impose the syntactic restriction that in every rule each variable has to occur in at least one positive atom of the rule's body. It turns out that both variants have the same expressive power. A discussion of these issues can be found in [3]. In the rest of this chapter we will assume that all quantifiers range over the active domain.

The widely used query language SQL combines and extends features of both, relational calculus and algebra.

Codd's following theorem summarizes one of the most fundamental results in database theory:

THEOREM 1.9 [26]
Relational calculus, relational algebra, and non-recursive rule-based queries have the same expressive power.

Furthermore, translations are effective, and the translation from relational calculus to relational algebra and vice versa can be done in polynomial time. In particular, it is always possible to *compile* a query expressed in the relational calculus into an expression of the relational algebra. The latter can then be evaluated efficiently.

Query languages with (at least) the expressive power of the relational algebra or the relational calculus are called *Codd-complete*.[¶] Languages which have exactly the same expressive power as the relational algebra are called *Codd-equivalent.*

It should be noted that *conjunctive queries* (cf. Section 1.3) correspond exactly to formulas of the relational calculus that use only conjunction and existential quantification, i.e., to formulas of the form $\exists \bar{x} \left(R_1(\bar{x}_1) \wedge \cdots \wedge R_\ell(\bar{x}_\ell) \right)$.

1.4.2   Limitations of the expressive power

Even though Codd-equivalent query languages like the relational algebra can express many everyday queries against a relational database, they still cannot express "everything". Theorem 1.9 is a key to understanding the precise expressive power of such languages: it relates them to the relational calculus, therefore allowing logic-based methods to prove that certain queries cannot be expressed.

In a nutshell, Codd-equivalent languages cannot count and cannot express recursion. E.g., they can neither ask "Which artist performed in more operas than any other artist?" nor "Is the total number of operas in the database even?". Furthermore, in a database that consists of a parent-child-relation, they cannot ask whether A is an ancestor of B.

[¶]Codd himself called these query languages *relationally complete* [26].

All these statements can be directly proved by *Ehrenfeucht-Fraïssé games*, as is explained, e.g., in [81]. Very often, however, the impossibility to express a certain query can also be concluded in a simpler way, either by using *0-1 laws* or by using *locality*. We present both notions next.

0-1 laws. Let q be a Boolean query and let σ be its schema, i.e., the set of relation names it mentions. For the moment, we consider only databases with schema σ whose active domain is an initial segment $\{1,\ldots,n\}$ of the natural numbers. We are interested in the ratio of databases of size n on which q yields the answer "yes" compared to *all* databases of size n, when n approaches infinity. More precisely, we denote by $\mu_n(q)$ the number of database instances over σ with active domain $\{1,\ldots,n\}$ on which q evaluates to "yes", divided by the number of all databases with schema σ and active domain $\{1,\ldots,n\}$. A Boolean query q is *almost surely true* (respectively, *almost surely false*) if the limit $\mu(q) := \lim_{n\to\infty}\mu_n(q)$ exists and is 1 (respectively, 0).

For instance, it is not hard to see that for query (8) from Section 1.2.2, i.e., for the query EvenOperas asking whether the number of operas in the database is even, $\mu(\text{EvenOperas}) = 1/2$.

A query language Q is said to *have the 0-1 law* if every Boolean query of Q that does not mention any constants from the domain of potential data values is almost surely true or almost surely false.

THEOREM 1.10 [44, 38]
Codd-equivalent query languages have the 0-1 law.

A simple consequence of this is that EvenOperas cannot be expressed by Codd-equivalent query languages. Also, many other counting queries q either have no limit $\mu(q)$ or a limit different from 0 and 1 and thus are not expressible by Codd-equivalent languages. Note, however, that counting "up to a constant threshold" is possible with Codd-complete languages; for example query (3), asking whether there is an artist starring in at least two theaters, can be expressed in the relational calculus via $\exists x \exists y_1 \exists z_1 \exists y_2 \exists z_2 \big(\text{Cast}(z_1,y_1,x) \wedge$

$\text{Cast}(z_2,y_2,x) \wedge \neg z_1 {=} z_2\big)$.

Locality. Unfortunately, the 0-1- law is not a very natural tool in the presence of non-Boolean queries. However, *locality* arguments can often be used to extend inexpressibility results to non-Boolean queries. In a nutshell, a query language is called local if it cannot express queries that depend on an unbounded number of tuples "connecting" one data item with another one (like, for example, the query Ancestor(x,y) which asks whether person x is an ancestor of person y in a parent-child database).

To be more precise, the *Gaifman-graph* G_D of a database D is the undirected graph whose vertices are the elements of the active domain of the database, and there is an edge between two vertices whenever they appear in a tuple of a relation of D. The *distance* between two elements of D is their distance in G_D. For any number k, the *k-neighborhood* $N_k^D(\bar{a})$ of a tuple $\bar{a}$ of elements is the sub-database induced by all elements of D that have distance at most k from some element of $\bar{a}$. E.g., the k-neighborhood of a person in the parent-child database consists of all persons to which he or she is related by at most k parent-child tuples and all tuples of the database containing only such persons.

A query q is called *Gaifman-local* if there exists a number k such that for any database D and any tuples $\bar{a}$ and $\bar{b}$ of D (of the right arity for q), if whenever $N_k^D(\bar{a})$ is isomorphic to $N_k^D(\bar{b})$ then $\bar{a} \in q(D)$ iff $\bar{b} \in q(D)$.

Obviously, there is no such k for the ANCESTOR query, i.e., the ANCESTOR query is *not* Gaifman-local. The following theorem therefore tells us that Codd-equivalent languages cannot express this query:

THEOREM 1.11 [41, 62]
Codd-equivalent languages can only express Gaifman-local queries.

Theorem 1.11 can be considered a formalization of the intuitive statement that Codd-equivalent languages lack recursion.

There are other notions of locality like *Hanf-locality* and *bounded number of degrees property* that hold for Codd-equivalent languages and some of their extensions. In particular, Hanf-locality can be used to prove that certain Boolean queries respecting the 0-1 law (like connectivity of a graph) cannot be expressed by Codd-equivalent languages. We refer the interested reader to [81, 80] and the references therein.

1.4.3 Complexity

Evaluation. Each operator of the relational algebra can be evaluated in a straightforward way. For example, the naive processing of a join operation $R_1 \bowtie R_2$ roughly requires $|R_1| \cdot |R_2|$ steps since each tuple in R_1 is combined with all tuples in R_2. In general, the evaluation of a query which involves intermediate results of arity k can be evaluated on a database D in time $O(|\mathrm{adom}(D)|^k)$. Likewise, formulas of the relational calculus can be evaluated by essentially turning each quantifier into a `FOR`-loop, resulting in a similar complexity.

Parts (b) and (c) of the following theorem show that, in the worst case, this upper bound cannot be significantly improved unless some widely believed complexity theoretic assumptions fail. Recall that we denote the relational calculus by FO because it is based on first-order logic formulas. Although formulated for FO, the following theorem also holds for the relational algebra.

THEOREM 1.12

(a) *The data complexity of* Eval(FO) *is in* AC^0 *[65] and thus, in particular, in* LogSpace.

(b) *The combined complexity of* Eval(FO) *is* PSpace*-complete [106, 99].*

(c) *The parameterized complexity of* Eval(FO) *(with the size of the query as parameter) is* AW[∗]*-complete [36].*

The results above show that evaluating Codd-equivalent queries in a scenario where the query is not fixed is rather difficult in general. Just as in the case of Theorem 1.2 for conjunctive queries, this seems to contradict the empirical experience that SQL queries can usually be evaluated reasonably fast. The explanation for this has several facets:

First of all, Theorem 1.2 and Theorem 1.12 talk about *worst cases*. The queries constructed in the proofs are of a very complicated structure that usually does not occur in practice. This observation is the starting point for many investigations on how the structure of queries influences the evaluation complexity. We will come back to this issue in Section 1.5.

A second aspect is that in practice, queries are rather small whereas databases are large. Thus, data complexity seems to be a better measure than combined complexity — and Theorem 1.12 (a) tells us that the data complexity is very low.

Furthermore, the structure of the database can have an impact on the complexity, and usually databases are not "arbitrarily complicated". In fact, the parameterized complexity of Eval(FO) gets feasible when attention is restricted to certain classes of database instances. For example, in [32] it is shown that over classes of databases that *locally exclude a minor* (see [32] for the definition), the parameterized complexity of Eval(FO) is fixed parameter tractable, i.e., belongs to the complexity class FPT. This result subsumes most of the previously known fixed parameter tractability results for query evaluation; see [54] for a survey. Nevertheless, restricting the structure of databases does not help to improve the evaluation complexity in the setting of Theorem 1.12 (b) since the combined complexity of Eval(FO) is PSpace-complete already on databases with only two data items.

Finally, the massive amounts of data handled by database systems usually reside in external memory. When processing such data, the input/output communication between fast internal memory and slower external memory is a major performance bottleneck: during the time required for a single random access to external memory, a huge number of computation steps could be performed on data present in internal memory. Indeed, modern database technology uses clever heuristics to minimize the costs caused by accesses to external memory (cf., e.g., [108, 90]). Classical complexity classes such as PTime and NP, however, measure complexity only by counting the

total number of computation steps, but do not take into account the existence of different storage media. In recent years, machine models that distinguish between external memory and internal memory have also been proposed and studied (for an overview, see the surveys [108, 96] and the references therein).

Static analysis.   In general, static analysis for Codd-complete languages is impossible:

THEOREM 1.13 [101]
The satisfiability problem Sat(FO) *is undecidable.*

   As a consequence, one immediately obtains that also the equivalence problem Equiv(FO) and the containment problem Cont(FO) are undecidable. The next section presents a couple of restrictions of FO for which static analysis is decidable.

1.5   Towards lower complexity: restricted query languages

In this section we revisit the idea that queries in practice, even if they go beyond conjunctive queries, are not arbitrarily complicated. We consider restrictions of the relational calculus FO (and other Codd-equivalent languages) which are not Codd-complete, but for which static analysis is decidable and the combined complexity of query evaluation is considerably lower than that of full FO.

   First, we concentrate on simple extensions of conjunctive queries by adding either union or inequalities. Afterwards, we consider a restriction of the relational calculus in which the number of variables is bounded. Finally, we consider a variant of the relational algebra in which the use of joins is restricted. We will see that in the latter case the resulting query language corresponds to a variant of the relational calculus in which the use of quantifiers is restricted, and also to a variant of the non-recursive rule-based queries in which single rules are based on acyclic conjunctive queries.

1.5.1   Extensions of conjunctive queries

There are several simple ways to extend the rule-based approach of CQ in order to gain more expressive power. For instance one could allow other kinds of atoms in the body of a rule, typically atoms of the form $x \neq y$. One could also consider defining a query using several rules instead of just a single rule. We denote by CQ($\neq$) the extension of CQ allowing inequality

atoms in the body of a rule. The class UCQ of *unions of conjunctive queries* is the extension of CQ allowing a finite number of rules in the definition of a query. We use UCQ($\neq$) to denote the combination of the two extensions. For instance, query (3) of Section 1.2.2 can be expressed in CQ($\neq$), and query (2) can be expressed in UCQ, while none of them is expressible in CQ.

It is easy to see that UCQ, CQ($\neq$), and UCQ($\neq$) have exactly the same data complexity and combined complexity as CQ. The satisfiability problem Sat(CQ($\neq$)) (and therefore also Sat(UCQ($\neq$))) can be decided in polynomial time. The containment problem for the three languages is still decidable, but slightly more difficult than that of CQ. In the theorem below, Π_2^P refers to the second level of the polynomial time hierarchy (recall that (NP $\cup$ co-NP) $\subseteq$ $\Pi_2^P \subseteq$ PSpace).

THEOREM 1.14

(a) *If* $\mathbf{Q}$ *is one of the query languages* CQ($\neq$), *UCQ, UCQ($\neq$), then*

> *(i)* *The data complexity of* Eval($\mathbf{Q}$) *is in* AC^0 *and thus, in particular, in* LogSpace.

> *(ii)* *The combined complexity of* Eval($\mathbf{Q}$) *is* NP-*complete.*

(b) *The containment problem for* CQ($\neq$), UCQ($\neq$) *is* Π_2^P-*complete [93, 105]. The containment problem for* UCQ *is* NP-*complete.*

For more information on extensions of conjunctive queries see [73, 93, 105] and the references therein.

1.5.2 Bounding the number of variables

A natural way of restricting the relational calculus is to bound the number of variables used in formulas. For each number $k \geq 1$ let FO^k be the restriction of FO to formulas that use at most k variables. Note that variables may be re-quantified inside a formula.

It turns out that bounding the number of variables to k improves the evaluation complexity for every k but enables static analysis only for $k = 2$:

THEOREM 1.15

(a) *For each* $k \geq 2$, *the combined complexity of* Eval(FO^k) *is* PTime-*complete [107].*

(b) *The satisfiability problem* Sat(FO^2) *is decidable [84]. As a consequence,* Equiv(FO^2) *and* Cont(FO^2) *are also decidable.*

(c) For each $k \geq 3$, the satisfiability problem $\mathrm{Sat}(\mathrm{FO}^k)$ is undecidable[∥] [68, 59, 14].

As a consequence, also $\mathrm{Equiv}(\mathrm{FO}^k)$ and $\mathrm{Cont}(\mathrm{FO}^k)$ are undecidable.

In the same way as full FO, the k-variable fragment FO^k also has a rule-based counterpart, the so-called *NRSD-programs of strict tree-width at most $(k-1)$*. This is a restriction of the non-recursive rule-based queries where the query graph of every single rule has a strict tree decomposition of width at most $(k-1)$; see [39] for details.

A survey on FO^k and related finite variable logics can be found in [49].

1.5.3   The semijoin algebra

When looking at the relational algebra one notices that, in terms of complexity of query evaluation, the most troublesome operation is join. The arity of $R_1 \bowtie R_2$ is usually larger than that of R_1 and R_2, and the size of $R_1 \bowtie R_2$ can be as large as the product of $|R_1|$ and $|R_2|$. An interesting restriction of the join operator $\bowtie$ is the semijoin operator $\ltimes$. Given two relations R and S, $R \ltimes S$ consists of all tuples of R that can be joined with some tuple in S in the sense of the operator $\bowtie$. In particular, $R \ltimes S$ has the same attribute names as R, and the result of $R \ltimes S$ always is a subset of R.

The *semijoin algebra* SA (cf. [76, 77]) is the variant of relational algebra where the join operator $\bowtie$ is replaced by the semijoin operator $\ltimes$. Strictly speaking, the semijoin algebra is defined in a slightly different framework where attributes do not have names and are addressed via column numbers instead. The reader might think of the semijoin algebra as being equipped with operators for selection, projection, renaming, union, difference, semijoin, and an additional operator with which columns of a relation can be duplicated (see [78] for details).

It is not difficult to see that for every database D and every semijoin algebra query q, the result $q(D)$ consists of so-called *stored tuples*, i.e., tuples that are obtained from tuples in D by projecting to and, possibly, duplicating some attributes. In particular, for each fixed semijoin algebra query q, the output size of q is at most linear in its input size, i.e., the number of tuples in the query result $q(D)$ is at most linear in the number n of tuples in the input database D. Of course, this also holds for all subexpressions of q. A remarkable result from [77] shows that the reverse is also true: Any relational algebra query all of whose subexpressions compute relations of size $O(n)$ is in fact expressible in the semijoin algebra. Furthermore, every query not expressible in the semijoin algebra has a subexpression that may produce results of size $\Omega(n^2)$.

[∥]Undecidability of (not necessarily finite) satisfiability of FO^3 follows from [68]. That finite satisfiability is undecidable as well follows from [59], see [14] for a discussion.

The complexity of query evaluation of the semijoin algebra is much lower than that of the relational algebra, and static analysis is decidable:

THEOREM 1.16 [76]

(a) Eval(SA) *can be solved in time* $O(k \cdot n)$, *where k denotes the size of the query and n denotes the size of the input database.*
In particular, the combined complexity of Eval(SA) *is in* PTime.

(b) *The satisfiability problem* Sat(SA) *is* ExpTime-*complete.*
As a consequence, Equiv(SA) *and* Cont(SA) *are also* ExpTime-*complete.*

The semijoin algebra cannot express all conjunctive queries, but at least it can express all *acyclic* conjunctive queries whose result is a set of stored tuples. Moreover, it can also express many natural non-conjunctive queries such as, e.g. query (7): the relational algebra formulation of this query given in Section 1.4.1 in fact belongs to the semijoin algebra.

In the same way as the relational algebra, the semijoin algebra also has a logical and a rule-based counterpart:

The logical counterpart is the *guarded fragment* of FO, denoted by GF (see [6]). GF is a fragment of FO where first-order quantifications have to be guarded by atomic formulas. More precisely, GF is defined as follows: All atomic formulas belong to GF. If φ and ψ belong to GF then also $\neg\varphi$, $(\varphi \wedge \psi)$ and $(\varphi \vee \psi)$ belong to GF. If α is an atomic formula and φ is a GF-formula whose free variables belong to the variables of α, then for every tuple $\bar{x}$ of variables, the formulas $\exists \bar{x}(\alpha \wedge \varphi)$ and $\forall \bar{x}(\alpha \rightarrow \varphi)$ belong to GF. A GF-formula is called *strictly guarded* if it either has no free variable or it is of one of the forms $(\alpha \wedge \varphi)$ and $\exists \bar{x}(\alpha \wedge \varphi)$, where α is an atomic formula and φ is a GF-formula whose free variables belong to the variables of α. We say that a GF-formula is *guarded by stored tuples* if it is a disjunction of strictly guarded GF-formulas.

The rule-based counterpart of SA is based on the restriction of the non-recursive rule-based queries where every single rule is a variant of an acyclic conjunctive query in which also *negated* atoms may occur. The resulting queries are called *strictly acyclic NRSD-programs*; for the precise definition we refer to [39]. A slightly different rule-based characterization is proposed in [46], the so-called *recursion-free Datalog LITE*.

The following theorem summarizes the relation between SA, GF, and their rule-based counterparts.

THEOREM 1.17

(a) SA *has the same expressive power as the class of* GF*-formulas that are guarded by stored tuples [76].*

(b) The class of GF*-formulas that are guarded by stored tuples has the same expressive power as the strictly acyclic NRSD-programs [39].*

(c) Recursion-free Datalog LITE can express the same Boolean queries as sentences in the guarded fragment GF *[46].*

The translations between SA, GF, and the rule-based languages that are provided by the proof of the Theorem 1.17 are effective. The translations from GF to SA, to strictly acyclic NRSD-programs, and to recursion-free Datalog LITE, respectively, can be done in polynomial time. The translations in the opposite directions are a bit more involved.**

1.6 Towards more expressiveness: recursion and counting

As pointed out in Section 1.4.2, Codd-equivalent query languages neither support recursion nor counting. Having these limitations in mind, it is natural to extend Codd-equivalent languages by recursion or counting capabilities while maintaining as much of the desirable algorithmic properties as possible. The query language SQL, in particular, contains constructs for counting and, starting with the SQL:1999 standard, also for expressing queries involving recursion. Corresponding extensions of the relational algebra and relational calculus have been investigated in [80].

In this section, we consider rule-based query languages with recursion and logics that are enhanced by fixpoint operators and counting. Furthermore, we discuss the possibility of finding a query language that precisely expresses the polynomial-time computable queries. Finally, we briefly consider more expressive languages that are capable of defining also queries of higher complexity.

1.6.1 Rule-based queries with recursion

The query language *datalog* is a rule-based language which allows more than one rule, recursion, but no negation. A simple example is the following program which defines in a parent-child database a relation $\textsc{Ancestor}(x, y)$

**The translations from [76, 39, 46] induce an exponential blow-up in terms of the size of the queries. It remains open if more efficient translations exist.

consisting of all pairs (a,b) for which a is an ancestor of b:

$$\text{ANCESTOR}(x,y) \; : - \; \text{PARENT}(x,y)$$
$$\text{ANCESTOR}(x,y) \; : - \; \text{PARENT}(x,z), \; \text{ANCESTOR}(z,y).$$

More formally, a *datalog program* over a database schema σ consists of finitely many rules of the form $Q(\bar{x}) : -R_1(\bar{t}_1), \cdots, R_\ell(\bar{t}_\ell)$ where the relation symbol Q does not belong to σ, and each variable in $\bar{x}$ occurs in at least one of the tuples $\bar{t}_1, \ldots, \bar{t}_\ell$. Relation symbols occurring in the head of some rule are called *intensional* relation symbols, whereas the symbols in σ are called *extensional* relation symbols. The body of each rule of a datalog program consists of atoms with intensional or extensional relation symbols. The *schema* σ_P of a datalog program P consists the symbols in σ and the intensional relation symbols of P.

Given a database D of schema σ, a datalog program P is evaluated as follows: start with *empty* intensional relations and proceed step-wise, by adding tuples that satisfy a rule of the program, until nothing changes. Formally, the semantics of a datalog program P can be defined in various equivalent ways. One possibility is to associate with P an operator T_P, the so-called *immediate consequence operator*, which maps a database E of schema σ_P to a database $T_P(E)$ of the same schema: Extensional relations R remain unchanged, i.e., $R^{T_P(E)} = R^E$. For each intensional relation symbol Q, the relation $Q^{T_P(E)}$ is obtained as follows: take all rules of P whose head contains Q, view each of these rules as a conjunctive query, and let $Q^{T_P(E)}$ be the union of the results of these conjunctive queries when applied to database E.

The *result* of P when applied to a database D of schema σ is the database $P(D)$ of schema σ_P obtained as follows: let D^0 be the extension of D to schema σ_P where all intensional relations are empty, and repeatedly apply the operator T_P to obtain databases $D^1 := T_P(D^0)$, $D^2 := T_P(D^1)$, $D^3 := T_P(D^2)$, etc. I.e., D^i is obtained by starting with D^0 and applying the operator T_P for i times. As T_P is a *monotone* operator[††], the sequence of the D^i is increasing, i.e., $D^0 \subseteq D^1 \subseteq D^2 \subseteq \cdots$. Since the active domain of each D^i consists of constants occurring in P and of elements from the active domain of the original (finite) database D, a *fixpoint* will be reached eventually, i.e., there exists a number j such that $D^j = D^{j+1}$. The result of P on D is defined as $P(D) := D^j$. It is not difficult to see that j is of size polynomial in the size of D and that $P(D)$ can be computed in time polynomial in the size of D (with the exponent depending on the particular datalog program P). Furthermore, $P(D)$ is actually the *least* fixpoint of T_P that contains D.

A *datalog query* is a datalog program together with a designated intensional relation symbol which specifies the relation defined by the query. In the following we write "Datalog" to denote the class of all datalog queries.

[††]in the sense that for all databases E and E' of schema σ_P, $E \subseteq E'$ implies that $T_P(E) \subseteq T_P(E')$

Concerning limitations of the expressive power it should be noted that, similarly as CQ (see Theorem 1.1), Datalog can define only *monotone* queries (this immediately follows from the fact that T_P is a monotone operator). Thus, the expressive power of Datalog is incomparable to the expressive power of Codd-equivalent query languages: On the one hand, there are simple non-monotone queries such as, e.g., query (6) from Section 1.2.2, that are expressible in the relational calculus FO but not in Datalog. On the other hand, there are recursive queries such as, e.g., the ANCESTOR query, that can be expressed in Datalog but not in FO.

There are many results on algorithmic properties of Datalog. We only mention the main results here; for a survey the reader is referred to [31]. In terms of query evaluation, the following holds:

THEOREM 1.18

(a) *The data complexity of* Eval(Datalog) *is* PTime-*complete (implicit in [106, 64]).*

(b) *The combined complexity of* Eval(Datalog) *is* ExpTime-*complete (implicit in [106]).*

Concerning the worst case complexity of Eval(Datalog), it is known (even without relying on any complexity theoretic assumption) that the exponential dependence on the size of the input query cannot be avoided: there exists a sequence of (Boolean) Datalog queries q_k of size polynomial in k, such that $q_k(D)$ can be computed in time $|\text{adom}(D)|^k$ but not in time $|\text{adom}(D)|^{k-1}$. The proof even holds for a suitable fixed database schema σ; the arity of the intensional relation symbols of q_k, however, has to grow with increasing k. When restricting attention to relations of a fixed arity, the parameterized complexity of Eval(Datalog) is known to be W[1]-complete, i.e., the same as for conjunctive queries (cf. Theorem 1.2 (c)). Details can be found in [88].

In terms of static complexity the following holds:

THEOREM 1.19

(a) *The satisfiability problem* Sat(Datalog) *is decidable.*

(b) *The equivalence problem* Equiv(Datalog) *and the containment problem* Cont(Datalog) *are undecidable [98].*

The proof of (a) is easy (see [3] for details). Concerning equivalence and containment, it should be noted that they are undecidable for datalog *queries*. Their "uniform" variants for datalog *programs*, asking whether *all* intensional relations defined by one program are equivalent to, respectively, included in the corresponding intensional relations of another datalog program, are

decidable [92]. Furthermore, query containment becomes decidable if one of the two involved queries is non-recursive [92, 22].

Apart from equivalence and containment, another key problem relevant for static analysis of Datalog is the *boundedness problem*, asking for a given datalog query q whether or not the recursion depth necessary for evaluating q is bounded by a number that only depends on the query but not on the input database. This problem is closely related to the problem whether a given datalog query is expressible in FO, i.e., without using recursion. It is known that the boundedness problem is undecidable [42].

In the literature, various restrictions of Datalog have been identified for which the problems of boundedness, equivalence and containment are decidable (see, e.g., [28, 22] and the references therein). Also variants of Datalog that can be evaluated in linear time w.r.t. data complexity have been considered, e.g. Datalog LITE [46].

1.6.2 Relational calculus with fixpoints

In Section 1.6.1 we have seen how to extend conjunctive queries with a recursion mechanism via fixpoints. It is natural to consider a similar extension for the relational calculus.

Similarly to the immediate consequence operator T_P from Section 1.6.1, a formula of the relational calculus FO can define an operator on relations as follows: Let R be a relation symbol that is not present in the given database schema σ, and let k be the arity of R. Let $\varphi(\bar{x})$ be a formula with k free variables over the extended schema $\sigma \cup \{R\}$. Then, on any database D of schema σ, the formula $\varphi(\bar{x})$ defines an operator φ^D between k-ary relations over the active domain of D. Given a k-ary relation $\hat{R}$, $\varphi^D(\hat{R})$ is the result of the query $\varphi(\bar{x})$ when applied to the extension of D in which the relation symbol R is interpreted by the relation $\hat{R}$.

Analogously to the iterated application of the operator T_P in Section 1.6.1, we can now consider the iterated application of the operator φ^D, yielding a sequence of relations $R^0 := \emptyset, R^1 := \varphi^D(R^0), R^2 := \varphi^D(R^1), \ldots$, i.e., R^i is obtained by starting with the empty relation and applying the operator φ^D for i times. Note that, unlike with datalog, there exist FO-formulas φ (for example, the formula $\varphi(\bar{x}) := \neg R(\bar{x})$) for which the sequence $R^0, R^1, R^2, \ldots$ is not increasing and does not reach a fixpoint.

There are several natural ways of ensuring that only those operators are considered for which the sequence $R^0, R^1, R^2, \ldots$ is increasing and eventually reaches a fixpoint. In the following, we present two of them: monotone operators and inflationary operators.

Monotone operators and the logic LFP. It is easy to see that if φ^D is a *monotone* operator, then $R^0 \subseteq R^1 \subseteq R^2 \subseteq \cdots$, and a fixpoint will be reached eventually, i.e., there exists a j such that $R^j = R^{j+1}$. Furthermore, this fixpoint

is in fact the *least* fixpoint[‡‡] of the operator φ^D. So it would be natural to semantically restrict attention to those formulas φ for which the operator φ^D is monotone for all databases D. Unfortunately, this kind of monotonicity of a formula is undecidable (see e.g., the textbook [81]). Of course it makes little sense to define a query language based on an undecidable property of formulas, because then it is not even decidable if a given string belongs to the query language or not.

Fortunately, there is a second option which enforces monotonicity at the syntactic (rather than the semantic) level: restrict attention to those formulas φ in which R occurs only *positively*, i.e., within the scope of an even number of negations. It is straightforward to see that this implies that on every database D the operator φ^D is monotone. The opposite is of course not true but it turns out that any fixpoint obtained by a monotone formula can also be obtained by a positive formula [61]. Hence the syntactic restriction is harmless in terms of expressive power of the corresponding fixpoint logics.

This syntactic restriction leads to the *least fixpoint logic* LFP, which extends first-order logic FO by the following rule: If $\varphi(\bar{x})$ is an LFP-formula in which R occurs only positively, then $[\text{lfp}_{R,\bar{x}}\varphi](\bar{x})$ is also an LFP-formula. The semantics of this formula is as follows: The result of query $[\text{lfp}_{R,\bar{x}}\varphi](\bar{x})$ on a database D is the limit of the sequence $R^0 := \emptyset$, $R^1 := \varphi^D(R^0)$, $R^2 := \varphi^D(R^1)$, $R^3 := \varphi^D(R^2)$ etc., and thus the least fixpoint of φ^D.

For example, if $\varphi(x,y)$ is the formula $\text{Parent}(x,y) \vee \exists z(\text{Parent}(x,z) \wedge R(z,y))$, then $[\text{lfp}_{R,xy}\varphi](x,y)$ is an LFP-formula that defines the ancestor relation for parent-child databases.

Inflationary operators and the logic IFP.   Another way of ensuring that the considered sequence of relations $R^0, R^1, R^2, \ldots$ is increasing is to use instead of φ^D the operator I_φ^D which maps a relation $\hat{R}$ to the relation $I_\varphi^D(\hat{R}) := \hat{R} \cup \varphi^D(\hat{R})$. By definition, the sequence based on this operator, i.e., the sequence $R^0 := \emptyset$, $R^1 := I_\varphi^D(R^0)$, $R^2 := I_\varphi^D(R^1)$, $\ldots$ is increasing and thus eventually reaches a fixpoint (no matter what $\varphi(\bar{x})$ looks like). This fixpoint is called the *inflationary fixpoint of φ*. The *inflationary fixpoint logic* IFP extends first-order logic FO by the following rule: If $\varphi(\bar{x})$ is a IFP-formula, then $[\text{ifp}_{R,\bar{x}}\varphi](\bar{x})$ is also an IFP-formula. The semantics of this formula is as follows: The result of query $[\text{ifp}_{R,\bar{x}}\varphi](\bar{x})$ on a database D is the fixpoint reached by the sequence $R^0 := \emptyset$, $R^1 := I_\varphi^D(R^0)$, $R^2 := I_\varphi^D(R^1)$, $R^3 := I_\varphi^D(R^2)$, etc.

Complexity and expressive power of LFP and IFP.   From the definition of the logics it is not difficult to see that the data complexity of evaluating queries definable in LFP or IFP belongs to PTime. In fact, query evaluation of these languages has the following complexity:

[‡‡]i.e., $R^j \subseteq S$, for every relation S with $\varphi^D(S) = S$

THEOREM 1.20

(a) *The data complexity of* Eval(IFP) *and* Eval(LFP) *is* PTime-*complete [106, 64].*

(b) *The combined complexity of* Eval(IFP) *and* Eval(LFP) *is* ExpTime-*complete [106].*

Since LFP and IFP are extensions of FO, static analysis of these languages is impossible in general (cf. Theorem 1.13). Concerning the expressive power of LFP and IFP, the following is known:

THEOREM 1.21 *[61]*
IFP *can express exactly the same queries as* LFP.

Note that in the definition of the logics LFP and IFP, nesting of fixpoints is explicitly allowed. Restricting attention to formulas where just a single fixpoint operator may be applied to a first-order formula does not change the expressive power, since nested fixpoints can always be simulated by a single fixpoint (of potentially higher arity) [64].

It is also possible to get the same expressive power using an extension of Datalog with negation. Even though the definition of the semantics of this extension is not an obvious issue (see [3]), it turns out that with the so-called *well-founded semantics* it has the same expressive power as IFP and LFP [43].

Even though recursion adds a lot of expressive power to the relational calculus, it does not help to count. Indeed, the following holds:

THEOREM 1.22 *[13]*
LFP *and* IFP *have the 0-1 law.*

Consequently, e.g. the EvenOperas query, i.e., query (8) from Section 1.2.2, is not definable in LFP or IFP.

For more details on fixpoint logics the reader is referred to the textbooks [3, 81].

1.6.3 Relational calculus with counting

SQL has several numerical features and counting features which are actually used in practice much more often than the recursion mechanisms. In Section 1.4.2 we have seen, however, that the relational calculus basically cannot count: it even cannot express the query EvenOperas, asking whether the number of operas in the database is even.

The simplest way to extend the relational calculus with counting facilities is to explicitly include them in the syntax: We consider the extension FO+C

of FO with counting quantifiers. FO+C is a two-sorted logic with the second sort being the natural numbers. The formula

$$\exists i \ \Big(\ \exists j \ (i{=}j{+}j) \ \wedge \ \exists ! i \ y \ (\exists x \ \textsc{Operas}(x,y)) \ \Big)$$

is an example of a FO+C-formula which expresses the EVENOPERAS query. Specifically, the formula states that there is a number i which is even (since there exists an integer j with $i{=}j{+}j$) such that the number of names y of pieces listed in the OPERAS-relation is exactly i. This formula combines the three kinds of quantifiers allowed in FO+C: Apart from the usual quantifiers ranging over elements of the active domain, it uses quantifiers of the form $\exists i$ that range over natural numbers (to be precise, the variables ranging over natural numbers only take values which are at most the size of the active domain of the underlying database). Furthermore, if variable i is interpreted by a natural number n, then a formula of the form $\exists ! i \ y \ \varphi(y)$ expresses that there are exactly n distinct elements a (in the active domain of the underlying database) for which $\varphi(a)$ holds. Apart from these quantifiers, FO+C also contains arithmetic predicates such as the linear order $<$, the addition $+$, and the multiplication $\times$.

Since FO+C contains FO, static analysis is impossible in general (cf. Theorem 1.13). The complexity of query evaluation for FO+C is not much higher than that of the relational calculus:

THEOREM 1.23

(a) The data complexity of Eval(FO+C) *is in* TC^0 *[11] and thus, in particular, in* LogSpace*.*

(b) The combined complexity of Eval(FO+C) *is* PSpace-*complete.*

Concerning the proof of (b), the inclusion in PSpace is straightforward, and the PSpace-hardness immediately follows from Theorem 1.12 (b).

By definition, FO+C extends the relational calculus with facilities for counting and for doing arithmetic. However, FO+C fails to express simple recursive queries such as, e.g., the ANCESTOR query. Actually, it is still Gaifman-local [79]. For a more detailed overview of FO+C and related logics, we refer to [79, 95] and the references therein.

1.6.4 The quest for PTime

As we have seen in Section 1.6.2 and 1.6.3, recursion or counting can be added to FO while keeping the data complexity of query evaluation in polynomial time. Often, a query is considered *tractable* iff it can be evaluated in polynomial time w.r.t. data complexity. It would, of course, be desirable to have a query language that is capable of expressing exactly the tractable queries,

i.e., exactly those queries that can be evaluated in polynomial time w.r.t. data complexity. The quest for such a query language is often stated as "Is there a language capturing PTime?".

We have seen that both FO+C and IFP fail to capture all of PTime: for example, FO+C cannot express the ANCESTOR query and IFP cannot express the EVENOPERAS query, but both queries can easily be evaluated in time polynomial in the size of the underlying database. It thus is natural to wonder whether the combination of IFP and FO+C (as it accounts for counting and recursion at the same time) does capture PTime. Let IFP+C be the language extending FO with both, inflationary fixpoints and counting. IFP+C has several nice properties, e.g., it can express the EVENOPERAS query and the ANCESTOR query, and its data complexity of query evaluation belongs to PTime. However it fails to capture *all* PTime-computable queries:

THEOREM 1.24 [16]
There exists a query that can be evaluated in polynomial time (w.r.t. data complexity) but that is not expressible in IFP+C.

Nevertheless, when restricting attention to particular classes of databases, IFP+C and, in some cases, even IFP do capture all of PTime. The following theorem summarizes some results in this vein. Here, an *ordered database* is a database which contains a relation that is a linear order on the database's active domain.

THEOREM 1.25

(a) IFP *captures* PTime *on the class of all ordered databases [64, 106].*

(b) IFP+C *captures* PTime *on all classes of databases of bounded treewidth [56] and on the class of databases corresponding to planar graphs [50].*

Note that (a) implies that IFP and IFP+C have the same expressive power on the class of ordered databases. Specifically, they express exactly those queries that can be evaluated in polynomial time data complexity.

Several generalizations of Theorem 1.25 (to larger classes of databases) and Theorem 1.24 (to certain restricted classes of databases) are known; see e.g. [34, 30].

The question of whether there is a query language or logic capturing PTime originated in the work of Chandra and Harel [18, 19] and is considered one of the main challenging open problems in database theory and finite model theory. The logic must be *reasonable* in a sense defined by Gurevich [60] (in particular it must have an effective syntax). More details on the "quest for PTime" can be found in [55, 86] and the references given therein.

1.6.5   Beyond PTime

All the query languages considered so far in this chapter are capable of expressing only queries that can be evaluated in polynomial time w.r.t. data complexity. Sometimes, however, there is a need for query languages that can define also more complex queries.

A simple way of extending the expressive power of least fixpoint logic or inflationary fixpoint logic is to drop the requirement that the sequence $R^0, R^1, R^2, \ldots$ is monotone: Similarly to the definition of LFP, we consider the sequence obtained by iterated application of the operator φ^D, starting with $R^0 := \emptyset$. Now, however, φ may be an arbitrary formula such that the sequence $R^0, R^1, R^2, \ldots$ is not necessarily increasing. Thus, depending on the particular formula φ and the database D at hand, the sequence may or may not reach a fixpoint. If no fixpoint is reached, the *partial fixpoint* is, by convention, defined to be the empty relation $\emptyset$. This leads to the *partial fixpoint logic* PFP which extends first-order logic FO by the following rule: If $\varphi(\bar{x})$ is a PFP-formula, then $[\text{pfp}_{R,\bar{x}}\varphi](\bar{x})$ is also a PFP-formula. The semantics of this formula is as follows: The result of query $[\text{pfp}_{R,\bar{x}}\varphi](\bar{x})$ on a database D is the fixpoint of the sequence $R^0 := \emptyset$, $R^1 := \varphi^D(R^0)$, $R^2 := \varphi^D(R^1)$, $R^3 := \varphi^D(R^2)$, etc., if such a fixpoint exists; otherwise, it is defined to be the empty relation $\emptyset$.

Notice that all the intermediate relations R^i are of size polynomial in the size of the database. Furthermore, convergence (respectively, non-convergence) to a fixpoint can be detected in PSpace. Altogether, this leads to an algorithm for evaluating a PFP-query in space polynomial in the size of the underlying database (where the exponent depends on the particular PFP-query).

Some more results on PFP are summarized in the following theorem:

THEOREM 1.26

(a) The data complexity of Eval(PFP) *is* PSpace-*complete [106].*

(b) PFP *has the 0-1 law [74].*

(c) PFP *captures* PSpace *on the class of all ordered databases [106].*

(d) IFP *has the same expressive power as* PFP *if, and only if,* PTime = PSpace *[5].*

It is straightforward to see that all IFP-queries can be expressed in PFP, i.e., PFP has at least the expressive power of IFP. Part (d) of Theorem 1.26 tells us that showing that the expressive power of PFP is strictly larger than that of IFP (on the class of arbitrary databases) is no easier than showing the (widely believed but, up to date, unproven) complexity theoretic assumption that PTime ≠ PSpace. On the other hand, part (b) implies that the EvenOperas query, i.e., query (8) from Section 1.2.2, cannot be expressed in PFP. This query, however, can easily be evaluated in polynomial time. Thus, on the class of

arbitrary databases, PFP is not capable of expressing all tractable queries, let alone, all PSpace computable queries. Thus, PFP does not capture PSpace on the class of *all* databases. When restricting attention to the class of all *ordered* databases, however, Theorem 1.26 (c) tells us that PFP precisely captures PSpace.

To conclude this section let us briefly consider a different (and widely used) way of extending the expressive power of query languages: Most database systems allow embedding of SQL queries into more powerful programming languages like C++ and java. This gives the user the capabilities to express *arbitrary* queries. The price to pay, however, is twofold: First of all, no optimization is performed except for, possibly, the SQL parts of the query. Furthermore, embedding a query language in a programming language implies that it is possible to specify queries that are not computable, i.e., queries for which there exists no evaluation algorithm that, on any input database D, stops after a finite number of steps and outputs the query result.

Several formalizations of the approach of embedding a query language into a programming language have been considered in the literature. We refer to chapter 18 of [3] for a discussion on such and other highly expressive languages.

1.7 Towards more flexibility: query languages for other data models

Throughout the previous sections we restricted attention to the relational data model. In the present section we conclude with a brief discussion of some other major data models.

Set semantics vs. bag semantics. The relational model is based on the so-called *set semantics*: no relation can have two (or more) identical tuples. During the evaluation of a query, however, it is often the case that identical tuples are generated, for instance when projecting a relation on one of its attributes. Eliminating duplicates is a costly operation as it often requires to sort the tuples in order to identify duplicates. Moreover, in some applications, counting the numbers of duplicates may be desirable, e.g. for the query "How many shows are performed in the MET each year?". Therefore, in most relational database systems duplicates are not eliminated unless explicitly requested by the query (this is the role of the SELECT DISTINCT construct of SQL).

Instead of the set semantics (considered throughout the previous sections of this chapter), the relational model can also be equipped with a *bag semantics*. With bag semantics, the number of occurrences of a tuple in the output of a

CQ query corresponds to the number of variable assignments satisfying the query.

With respect to query evaluation, the change from set semantics to bag semantics does not cause a considerable change in complexity (at least not for the "worst-case" analysis), since the complexity of intermediate sorting steps usually is dominated by the complexity of evaluating joins.

However, going from set semantics to bag semantics does change the complexity of static analysis, since the containment problem and the equivalence problem must take into account the number of occurrences of tuples in the query result. E.g., two queries are equivalent iff they produce the same tuples with the same multiplicities. This makes a dramatic difference. Indeed, considering the class CQ of conjunctive queries under bag semantics, it is known that the equivalence problem Equiv(CQ) is as hard as testing graph isomorphism (and hence it is in NP) [21]. For the containment problem under bag semantics, however, it is still an open question whether Cont(CQ) is decidable.

For the class UCQ of unions of conjunctive queries and for the extension CQ($\neq$) of conjunctive queries where also inequalities are allowed as atoms, the containment problems Cont(UCQ) and Cont(CQ($\neq$)) become undecidable [66, 67]. Recall from Theorem 1.14 that under set semantics these two problems are decidable. Concerning, however, the equivalence problem under bag semantics, it is known that Equiv(CQ($\neq$)) is decidable and belongs to PSpace [27].

The nested relational model.   A further limitation of the relational model is given by the so-called *first normal form* which restricts attribute values to being atomic. In this sense, relations in the relational model are *flat*. The *nested relational model* (or *complex value model*) contains nested type constructors which allow to build nested relations from atomic types by using tuple constructors and set constructors. Apart from suitable generalizations of the operators of the (flat) relational algebra, it has operators for nesting and unnesting relations. An equivalence between the logical calculus and the algebra can be established just as in the flat case [89, 1]. Interestingly, the nested operators do not add any expressive power to the relational algebra for flat queries on flat databases [89]. This result was generalized in [110] for non-flat queries. On the other hand, adding a powerset operator to the nested relational algebra strongly extends its expressive power, e.g., it allows to express the ANCESTOR query [1]. Apart from the references mentioned above, we refer to [3, 15] for more detailed information.

Object-oriented databases.   Object-oriented databases further extend the nested relational model towards the object oriented paradigm. Each object, or entity, is given a unique identifier (OID). In the relational model this would correspond to adding an extra "ID" attribute to each relation and requiring

that this attribute be a *key*. The model also allows OID as a possible attribute for an object. Query languages can be derived from those for nested relations, cf., for example, the language OQL [24], which was implemented in the O_2 database system [10]. A logical foundation for the object-oriented model was presented in [71]. For further information we refer to [17, 3, 104, 4, 35].

Constraint databases. Another feature of the relational model is that all relations are *finite*. There are many applications where this restriction can be seen as a limitation, for instance in spatial databases where a relation may correspond to a set of points in the plane. The *constraint database* framework is an elegant way of extending the relational model beyond finite relations. The idea is to manipulate implicit presentations of relations instead of explicit presentations that list all tuples. This is done by first specifying a logical framework, for instance first-order logic on the field of reals. Then, each (possibly infinite) relation is represented by a formula of the logical framework. In the case of first-order logic over the real field, this defines precisely the *semi-algebraic* relations. Then, depending of the logical framework and on the query language at hand, query evaluation can or cannot be performed effectively. Typically, if the logical framework is first-order logic and admits quantifier elimination, as in the real field, then first-order queries can be evaluated (actually in PTime data complexity for the case of the real field). Since the seminal paper on constraint databases [70], this area has generated a lot of theoretical work concerning various logical frameworks, and several prototypes have been developed. The main application is in spatio-temporal databases. For a short introduction to constraint databases we refer to [103]; a comprehensive survey is provided in the book [75].

The semi-structured data model. With the Internet, data is geographically distributed, and vast amounts of data are frequently exchanged between several database (and non-database) systems. The rigid structure of a relational database, enforced by its schema, makes it difficult to use the relational data model for transferring data from one site to another. To this end a new, more flexible model has been introduced: the *semi-structured* model (see [2] for a comprehensive overview). The general idea is that data is now "self-describing", i.e. its structure is part of the data. The most widespread implementation of the semi-structured model is the Extensible Markup Language (XML), specified by the World Wide Web Consortium. An XML document is a well-formed nested sequence of opening and closing *tags* in between which text and data values can be found. The tag structure can be seen as a labeled tree; it provides the "structure" of the document. A crucial difference with the relational setting is that data is now queried also by its position in the (possibly deeply nested) document tree. Hence, XML query languages must combine navigation in the document tree with other, more classical, database functionalities such as joins. Many query languages for XML have

been proposed. They are designed for differing purposes, e.g., navigating to (or extracting) positions in a document, transforming documents into new documents, and posing Boolean queries about documents. The most important XML query languages, XPath, XSLT and XQuery are maintained by the World Wide Web Consortium and are still under development, see `http://www.w3.org/XML/` for up-to-date information.

The theoretical foundations build upon concepts from automata theory and mathematical logic, among them tree automata, monadic second-order logic, and logics with navigating features similar to those in temporal logic. For details see [82, 72, 97] and the references therein.

The data stream model. The data stream model considers data that is not stored but, instead, arrives in multiple, continuous, rapid, and time-varying streams. Typical application areas for which data stream processing is relevant are, e.g., IP network traffic analysis, mining text message streams, or processing data generated by sensor networks. In all these application areas it is not feasible to simply load the arriving data into a traditional relational database management system (DBMS) and query it there. Instead, the streaming data has to be processed on-the-fly by using only a limited amount of memory. Instead of the precise query answers provided by traditional DBMS, queries against data streams are often evaluated by randomized algorithms that produce approximate solutions. Systems-oriented overviews of query languages for data streams and general purpose data stream mangagement systems (DSMS) can be found in [7, 100]. A survey of machine models and lower bounds for stream-based query processing is given in [96]. For an introduction to efficient algorithms for data stream processing see [85].

Many other data models have been considered in the literature, each with its own query languages, and it seems that there will always be new data models and database applications. The area of query languages will thus be evolving in the foreseeable future and there remain a lot of challenges for research.

References

[1] Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4(4):727–794, 1995.

[2] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[4] Serge Abiteboul and Paris C. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45(5):798–842, 1998.

[5] Serge Abiteboul and Victor Vianu. Computing with first-order logic. *Journal of Computer and System Sciences*, 50(2):309–335, 1995.

[6] Hajnal Andréka, István Németi, and Johan van Benthem. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 1–16, 2002.

[8] Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *Proc. EACSL Intl. Conf. on Computer Science Logic (CSL)*, volume 4207 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 2006.

[9] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proc. EACSL Intl. Conf. on Computer Science Logic (CSL)*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer-Verlag, 2007.

[10] François Bancilhon, Claude Delobel, and Paris C. Kanellakis, editors. *Building an Object-Oriented Database System. The Story of O2*. Morgan Kaufmann, 1992.

[11] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On Uniformity within NC^1. *Journal of Computer and System Sciences*, 41(3):274–306, 1990.

[12] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, 1983.

[13] Andreas Blass, Yuri Gurevich, and Dexter Kozen. A zero-one law for logic with a fixed-point operator. *Information and Control*, 67(1–3):70–90, 1985.

[14] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.

[15] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[16] Jin-yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identifications. *Combinatorica*, 12(4):389–410, 1992.

[17] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, California, 1994.

[18] Ashok K. Chandra and David Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[19] Ashok K. Chandra and David Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25(1):99–128, 1982.

[20] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. ACM SIGACT Symp. on the Theory of Computing (STOC)*, pages 77–90, 1977.

[21] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *real* conjunctive queries. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 59–70, 1993.

[22] Surajit Chaudhuri and Moshe Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. *Journal of Computer and System Sciences*, 54(1):61–78, 1997.

[23] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211–229, 2000.

[24] Sophie Cluet. Designing OQL: Allowing objects to be queried. *Information Systems*, 23(5):279–305, 1998.

[25] Edgar F. Codd. A relational model of data for large shared data banks. *Communication of the ACM*, 13(6):377–387, 1970.

[26] Edgar F. Codd. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[27] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Deciding equivalences among conjunctive aggregate queries. *Journal of the ACM*, 54(2), 2007.

[28] Stavros S. Cosmadakis, Haim Gaifman, Paris C. Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs (preliminary report). In *Proc. ACM SIGACT Symp. on the Theory of Computing (STOC)*, pages 477–490, 1988.

[29] Bruno Courcelle. Linear delay enumeration and monadic second-order logic. To appear in *Discrete Applied Mathematics*, 2008.

[30] Elias Dahlhaus and Johann A. Makowsky. Query languages for hierarchic databases. *Inf. Comput.*, 101(1):1–32, 1992.

[31] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[32] Anuj Dawar, Martin Grohe, and Stephan Kreutzer. Locally excluding a minor. In *Proc. IEEE Conf. on Logic in Computer Science (LICS)*, pages 270–279, 2007.

[33] Anuj Dawar, Martin Grohe, Stephan Kreutzer, and Nicole Schweikardt. Model theory makes formulas large. In *Proc. Intl. Conf. on Algorithms, Languages and Programming (ICALP)*, volume 4596 of *Lecture Notes in Computer Science*, pages 913–924. Springer-Verlag, 2007.

[34] Anuj Dawar and David Richerby. The power of counting logics on restricted classes of finite structures. In *Proc. EACSL Intl. Conf. on Computer Science Logic (CSL)*, volume 4646 of *Lecture Notes in Computer Science*, pages 84–98. Springer-Verlag, 2007.

[35] Jan Van den Bussche, Dirk Van Gucht, Marc Andries, and Marc Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44(2):272–319, 1997.

[36] Rodney G. Downey, Michael R. Fellows, and Udayan Taylor. The parameterized complexity of relational database queries and an improved characterization of W[1]. In *Combinatorics, Complexity, and Logic, volume 39 of Proceedings of DMTCS*, pages 194–213. Springer-Verlag, 1996.

[37] Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. on Computational Logic*, 8(4), 2007.

[38] Ronald Fagin. Probabilities on finite models. *Journal of Symbolic Logic*, 41(1):50–58, 1976.

[39] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *Journal of the ACM*, 49(6):716–752, 2002.

[40] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer-Verlag, 2006.

[41] Haim Gaifman. On local and non-local properties. In J. Stern, editor, *Proceedings of the Herbrand Symposium, Logic Colloquium '81*, pages 105–135. North Holland, 1982.

[42] Haim Gaifman, Harry G. Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. *Journal of the ACM*, 40(3):683–713, 1993.

[43] Allen Van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 1–10, 1989.

[44] Y.V. Glebskii, D.I. Kogan, M.A. Liogon'kii, and V.A. Talanov. Range and degree of realizability of formulas in the restricted predicate calculus. *Kibernetika*, 2:17–28, 1969.

[45] Goran Gogic, Henry A. Kautz, Christos H. Papadimitriou, and Bart Selman. The comparative linguistics of knowledge representation. In *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 862–869, 1995.

[46] Georg Gottlob, Erich Grädel, and Helmut Veith. Datalog LITE: a deductive query language with linear time model checking. *ACM Trans. on Computational Logic*, 3(1):42–79, 2002.

[47] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM*, 48(3):431–498, 2001.

[48] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[49] Martin Grohe. Finite variable logics in descriptive complexity theory. *Bulletin of Symbolic Logic*, 4:345–398, 1998.

[50] Martin Grohe. Fixed-point logics on planar graphs. In *Proc. IEEE Conf. on Logic in Computer Science (LICS)*, pages 6–15, 1998.

[51] Martin Grohe. Generalized model-checking problems for first-order logic. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS'01)*, volume 2010 of *Lecture Notes in Computer Science*, pages 12–26. Springer-Verlag, 2001.

[52] Martin Grohe. The parameterized complexity of database queries. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 2–92, 2001.

[53] Martin Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM*, 54(1), 2007.

[54] Martin Grohe. Logic, graphs, and algorithms. In *Logic and Automata: History and Perspectives*, number 2 in Texts in Logic and Games, pages 357–422. Amsterdam University Press, 2007.

[55] Martin Grohe. The quest for a logic capturing PTIME. In *Proc. IEEE Conf. on Logic in Computer Science (LICS)*, pages 267–271, 2008.

[56] Martin Grohe and Julian Mariño. Definability and descriptive complexity on databases of bounded tree-width. In *Proc. of Intl. Conf. on Database Theory (ICDT)*, volume 1540 of *Lecture Notes in Computer Science*, pages 70–82. Springer-Verlag, 1999.

[57] Martin Grohe and Nicole Schweikardt. Comparing the succinctness of monadic query languages over finite trees. *RAIRO – Theoretical Informatics and Applications*, 38:343–373, 2004.

[58] Martin Grohe, Thomas Schwentick, and Luc Segoufin. When is the evaluation of conjunctive queries tractable? In *Proc. ACM SIGACT Symp. on the Theory of Computing (STOC)*, pages 657–666, 2001.

[59] Y. Gurevich and I. Koryakov. Remarks on Berger's paper on the domino problem. *Siberian Math. Journal*, 13:319–321, 1999. In Russian.

[60] Yuri Gurevich. Logic and the challenge of computer science. In E. Boerger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[61] Yuri Gurevich and Saharon Shelah. Fixed-point extensions of first order logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.

[62] Lauri Hella, Leonid Libkin, and Juha Nurmonen. Notions of locality and their logical characterizations over finite models. *Journal of Symbolic Logic*, 64:1751–1773, 1999.

[63] Richard Hull. Relative information capacity of simple relational database schemata. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 97–109, 1984.

[64] Neil Immerman. Relational queries computable in polynomial time. *Information and Control*, 68(1–3):86–104, 1986.

[65] Neil Immerman. Expressibility and parallel complexity. *SIAM Journal on Computing*, 18(3):625–638, 1989.

[66] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Trans. on Database Systems*, 20(3):288–324, 1995.

[67] T. S. Jayram, Phokion G. Kolaitis, and Erik Vee. The containment problem for REAL conjunctive queries with inequalities. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 80–89, 2006.

[68] A. Kahr, E. Moore, and H. Wang. Entscheidungsproblem reduced to the ∀∃∀ case. *Proc. Nat. Acad. Sci. USA*, 48:365–377, 1962. In Russian.

[69] Paris C. Kanellakis. Elements of relational database theory. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 17, pages 1073–1155. Elsevier Science Publishers, 1990.

[70] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 299–313, 1990.

[71] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[72] Nils Klarlund, Thomas Schwentick, and Dan Suciu. XML: Model, schemas, types, logics, and queries. In Jan Chomicki, Ron van der Meyden, and Gunter Saake, editors, *Logics for Emerging Applications of Databases*, pages 1–41. Springer-Verlag, 2003.

[73] Phokion G. Kolaitis, David L. Martin, and Madhukar N. Thakur. On the complexity of the containment problem for conjunctive queries with built-in predicates. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 197–204, 1998.

[74] Phokion G. Kolaitis and Moshe Y. Vardi. Infinitary logics and 0-1 laws. *Information and Computation*, 98(2):258–294, 1992.

[75] Gabriel M. Kuper, Leonid Libkin, and Jan Paredaens, editors. *Constraint Databases*. Springer-Verlag, 2000.

[76] Dirk Leinders, Maarten Marx, Jerzy Tyszkiewicz, and Jan Van den Bussche. The semijoin algebra and the guarded fragment. *Journal of Logic, Language and Information*, 14(3):331–343, 2005.

[77] Dirk Leinders and Jan Van den Bussche. On the complexity of division and set joins in the relational algebra. *Journal of Computer and System Sciences*, 73(4):538–549, 2007.

[78] Dirk Leinders and Jan Van den Bussche. Repetitions and permutations of columns in the semijoin algebra. *RAIRO – Theoretical Informatics and Applications*, 2008. To appear. DOI: 10.1051/ita:2008023.

[79] Leonid Libkin. Logics with counting and local properties. *ACM Trans. on Computational Logic*, 1(1):33–59, 2000.

[80] Leonid Libkin. Expressive power of SQL. *Theoretical Computer Science*, 296:379–404, 2003.

[81] Leonid Libkin. *Elements of Finite Model Theory*. Springer-Verlag, 2004.

[82] Leonid Libkin. Logics for unranked trees: an overview. *Logical Methods in Computer Science*, 2(3), 2006.

[83] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[84] M. Mortimer. On languages with two variables. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 21(8):135–140, 1975.

[85] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[86] Alan Nash, Jeffrey B. Remmel, and Victor Vianu. PTIME queries revisited. In *Proc. of Intl. Conf. on Database Theory (ICDT)*, volume 3363 of *Lecture Notes in Computer Science*, pages 274–288. Springer-Verlag, 2005.

[87] Christos H. Papadimitriou. The complexity of knowledge representation. In *IEEE Conference on Computational Complexity*, pages 244–248, 1996.

[88] Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. *Journal of Computer and System Sciences*, 58(3):407–427, 1999.

[89] Jan Paredaens and Dirk Van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 29–38, 1988.

[90] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

[91] Omer Reingold. Undirected ST-connectivity in log-space. In *Proc. ACM SIGACT Symp. on the Theory of Computing (STOC)*, pages 376–385, 2005.

[92] Yehoshua Sagiv. Optimizing datalog programs. In *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, 1988.

[93] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.

[94] Francesco Scarcello. Query answering exploiting structural properties. *SIGMOD Record*, 34(3):91–99, 2005.

[95] Nicole Schweikardt. Arithmetic, first-order logic, and counting quantifiers. *ACM Trans. on Computational Logic*, 6(3):634–671, 2005.

[96] Nicole Schweikardt. Machine models and lower bounds for query processing. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 41–52, 2007.

[97] Thomas Schwentick. Automata for XML – a survey. *Journal of Computer and System Sciences*, 73(3):289–315, 2007.

[98] Oded Shmueli. Decidability and expressiveness of logic queries. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 237–249, 1987.

[99] Larry J. Stockmeyer. *The complexity of decision problems in automata and logic*. PhD thesis, MIT, 1974.

[100] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[101] Boris A. Trakhtenbrot. The impossibility of an algorithm for the decision problem for finite models. *Doklady Academii Nauk SSSR*, 70(569–572), 1950. In Russian; translated into English in Amer. Soc. Trans., Series 2, 23, 1963.

[102] Jeffrey D. Ullman. *Principles of database and knowledge-base systems, Volume I.* Computer Science Press, Inc., New York, NY, USA, 1988.

[103] Jan Van den Bussche. Constraint databases: A tutorial introduction. *SIGMOD Record*, 29(3):44–51, 2000.

[104] Jan Van den Bussche, Dirk Van Gucht, Marc Andries, and Marc Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44(2):272–319, 1997.

[105] Ron van der Meyden. The complexity of querying indefinite data about linearly ordered domains. *Journal of Computer and System Sciences*, 54(1):113–135, 1997.

[106] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proc. ACM SIGACT Symp. on the Theory of Computing (STOC)*, pages 137–146, 1982.

[107] Moshe Y. Vardi. On the complexity of bounded-variable queries. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 266–276, 1995.

[108] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33:209–271, 2001.

[109] Heribert Vollmer. *Introduction to circuit complexity*. Springer-Verlag, 1999.

[110] Limsoon Wong. Normal forms and conservative extension properties for query languages over collection types. *Journal of Computer and System Sciences*, 52(3):495–505, 1996.

[111] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, pages 82–94. IEEE Press, 1981.