Chapter 10
Databases

Michael Benedikt and Pierre Senellart

Introduction: Two Views of Database Research

This chapter is about *database research* (or as we abbreviate *DBR*). To people outside of computer science – and perhaps to many within – it will be unclear what this term means. First of all, what is a "database"? Used generally, it could mean any collection of information. It is obvious that there are deep scientific issues involved in managing information. But information and data are very general notions. Doesn't much of computing deal with manipulating data or information? Isn't everything data? Clearly the databases that DBR deals with must be something more specific.

Database research takes as its subject something more specialized: *database management software*, a term which we will use interchangeably with *database management systems* (DBMSs or "database managers"). This refers to a class of software that has emerged to assist in *large-scale manipulation of information in a domain-independent way*. By *domain-independent*, we mean to contrast it with, say, a software package that calculates your taxes, or tools that help show your family tree – these are definitely processing data, but are manipulating it in ways that are very specific to one dataset (set of data). In contrast, there are many complex tasks that are associated with *storage*, *update*, and *querying* in a "generic sense". As early as 1962 software products emerged that were dedicated to providing support for these tasks. A particular concern was with performing them on large amounts of data. A current DBMS can filter (read and select) gigabytes of data in seconds, and can manage terabytes (10^{12} bytes) of data. In the last few decades a vibrant industry has sprung up around DMBS tools and tool suites. For example, personal computer users would know the DBMS Microsoft Access;

M. Benedikt (✉)
University of Oxford, Oxford, UK
e-mail: michael.benedikt@cs.ox.ac.uk

P. Senellart
Télécom ParisTech, Paris, France
e-mail: pierre.senellart@telecom-paristech.fr

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_10,

software developers would know the major commercial vendors, such as Oracle and IBM, along with open-source database management systems such as MySQL.

The relationship of generic database management products to end-to-end applications has varied over the years. Many companies advertise software that provides simply "generic database management". In other cases DBMS software is bundled either in application suites, or with e-commerce suites that also handle issues that have little to do with data. Similarly at runtime the relationship of a DBMS to the rest of the application could take many forms; there could be a dedicated database management process communicating with other software processes via a well-defined protocol. Alternatively, DBMS functionality could be available as libraries. Regardless of these "packaging" issues, database management software can be considered a separate "functional entity" within an application. A systems programmer requires special training to create it, wherever it sits. Someone (an application programmer or an end-user) must interface with it, and often in doing so must understand how it is engineered and how to tune it.

So there is a special kind of software called *database management software*. In a narrow sense, DBR refers to *the scientific study of this software*. But why should there be a field of database research? You may grant that database software is important. But there is no theory of tractors or knapsacks, despite their utility and fairly well-defined scope. In the software domain there is no field of "spreadsheet theory", or "word processor research", at least not one of comparable significance. Why does a DBMS warrant a separate research area?

We give two answers to this question. First of all, the design of a DBMS is complex and the approaches to creating a DBMS are stable enough and specific enough to the setting to be amenable to scientific study. Software of major database vendors runs to hundreds of thousands of lines of source code, and has evolved over a period of decades. Spreadsheets are also complex, but the tools and design principles used are either not stable enough, or not unique enough to the spreadsheet setting to sustain a separate discipline. Thus we can refine the definition of DBR as the study of the stable architectural and component construction of database management systems – the fundamental languages, algorithms and data structures used in these systems. *Database theory*, a subset of DBR, would then be the formalization and analysis of these languages and algorithms, e.g., semantics of the languages, upper and lower bounds on the time or space used in the algorithms. The emphasis on *stable* components explains why many features in a DBMS are not the subject of much research – there are comparatively few research papers about report generators, administrative interfaces, or format conversion for their lack of stability.

The complexity and uniqueness of software dedicated to data management gives one justification for DBR. However, a deeper motivation is that database management techniques and algorithms have become pervasive within computer science. Many of the features of modern database systems that we shall discuss in this chapter – *indexing*, *cost-based optimization*, *transaction management* – that were first developed in the context of database management systems have become essential components in related areas as well. The study of logic-based languages, which received its impetus from the success of relational databases,

has had impact on understanding the relationship between declarative specification and computation throughout computer science. Thus much of DBR is concerned with the application and expansion of "large-scale data management techniques" wherever they are or could be relevant in computer science. This "migration" accounts for a fact which will be fairly obvious to the reader of the proceedings of database conferences: *much of current DBR is not closely connected to current DBMS product lines at all*. Much of DBR involves languages that do not exist within current DBMSs, or features that go radically beyond what current systems offer. They may deal with proposed extensions of real languages, or modeling languages that are put forth as theoretical tools but which are unrealistic for practical use.

This phenomenon is not unique to DBR. A significant portion of programming language research investigates ideas for novel languages or language features, and security research often looks at the possible ways in which security might be achieved, regardless of their practicality. Similarly, DBR examines the ways in which computers *could* manage large quantities of information, rather than how they do manage it. Thus, much of DBR deals with "managing information" in a very wide sense. It concerns itself with broad questions: How can new information sets be defined from old ones? How could one describe relationships between datasets, and how could we specify the information that a user or program might want from a collection of structured information? What kinds of structure can one find in information? What does it mean for two sets of information to be the same? In DBR, these questions are dealt with from a computer science perspective: precise description languages are sought and algorithmic problems related to these description formalisms are investigated.

We see that DBR has a manifold structure; much of it revolves around existing database management systems, while another aspect revolves around techniques for data management in the wider sense: for use as a component within other software tools (e.g. Web search), for insight into other areas of computer science, and to explore the possibilities for managing information. The structure of this chapter will reflect this.

By way of a short introduction, we start by giving a bit of history of database management systems, leading up to the creation of *relational* database management systems, which are the most important class of DBMSs currently. We go on to describe the input languages and structure of a "traditional" (relational) DBMS. We then give a sample of DBR that is oriented towards improving the processing pipelines of existing DBMSs. In a subsequent section, we turn towards research on expanding the functionality of data management systems, covering several significant extensions, sometimes quite radical, that have been explored. In the process we will try to give some idea of how research on these new systems has been integrated into the standard feature set of commercial database systems, and the extent to which it has had influence in other parts of computer science.

Owing to page length limits, for many systems we shall only sketch their main features, giving references for details. We shall devote more space to the informal level and leave technical-level details to references cited. Furthermore, we

shall assume that the reader has some familiarity with such basic matters as formatting data on punched cards and magnetic tapes and with common database processes such as *report generators* (just what they sounds like), *sorting* (e.g., sequencing data, say alphabetically, according to some key fields in a table of data) and traditional business applications such as payroll files. These topics are intuitively comprehensible without explicit introductory exposition.

The Relational Model

The Path to Relational Databases

The growth of database management software is a story of the growth of abstraction in computing systems. As in many other areas of computing, in the beginning there were low-level concrete tasks that were carried out first by special-purpose hardware, then by special-purpose programs – for example, reading data from punched cards, performing a hard-coded calculation on the data and then generating results or *reports* in a custom format. Starting from the 1950s there were programs dedicated to processing payroll data, purchase orders, and other pieces of *structured data*. The software was tailored not only to the application at hand, but to the hardware and the input formats. Even the data records themselves, starting with punch cards and later magnetic tape, were often created on a per-task basis, with no sharing of data between applications.

At the end of the 1950s software emerged that abstracted some general functionality used in many data processing activities. Report generation and sorting were two areas of particular interest. This abstract software evolved into the *file management systems* of the 1960s. Although software was still often bundled with hardware, independent software vendors such as Informatics began to offer file-processing software in competition with mainframe vendors like IBM. This decoupling spurred interest in making data processing software independent from hardware, just as later on it would spur independence from the operating system. File managers regarded structured files – plain text files that made use of some set of delimiters – as the basic input abstraction. Here is a description of one system (quoted from Postley 1998).

> This program has been developed in response to a large number and wide variety of requests for reports consisting of selected information from magnetic tape files. These requests usually require the preparation of a new program or modification of an existing program. This program provides a more general solution to the problems of information retrieval and report generation. It combines four generalized capabilities. It can
>
> 1. Utilize any of a wide variety of tape formats
> 2. Make selections on the basis of complex criteria

3. Produce reports in a wide variety of list-type formats
4. Produce several reports on a single pass of a magnetic tape file. This can be done with no appreciable increase in retrieval time

> By providing such a system, it is expected that a reduction in programming and machine time will be realized. These savings, will, of course, be magnified for those retrieval/report generations of short production life and for those reports requiring frequent alterations in selection criteria or report format.

The emphasis even in these "generalized" systems was on batch processing, i.e., on entire files. With the move from magnetic tape to disk storage in the early 1960s, the possibility of querying on-demand emerged. At the same time, the notion that centrally managed data would be a way of radically increasing business productivity became popular within corporate management circles (Haigh 2006). The vision was that managers would have a global integrated view of their business, being able to answer questions "instantly" that would have previously taken hours or even weeks of manual work. The development of advanced decision support and business analysis tools that would realize this vision came much later in the evolution of database systems. But the possibility of such systems inspired corporations to invest more heavily in data processing technology, and gave added impetus to the development of a flexible query language.

Throughout the 1960s systems were developed that had many of the features of modern DBMSs, including some ability to perform querying of shared data and to concurrently process updates sent from multiple processes. IBM's IMS (Patrick 2009) and General Electric's IDS, the latter created by Turing Award winner Charles Bachman, introduced more general procedures for defining data, the precursors of modern data definition languages. While prior systems had left much of the responsibility for management of concurrent updates to application programmers, systems such as IMS managed them transparently. Still, the emphasis was on batch mode or on a fixed set of queries. The view of data provided by IMS upon its initial product release in the late 1960s is described as follows (McGee 2009):

> The data model provided by the initial release of IMS was Data Language/1 (DL/1). In this model, databases consisted of records, records were hierarchic structures of segments, and segments were sets of fields stored in consecutive bytes. One field in a record root segment could be designated the record key. The program's interface to IMS provided calls to access records sequentially or by key; to navigate to segments within the record; and to insert, replace, and delete records.

Although in retrospect the convergence towards the current notion of *data management functionality* is clear, as the 1960s ended, alternative visions of the future were available. On the one hand, overlapping functionality in database systems was incorporated in general-purpose programming languages such as RPG and COBOL. These included querying and data definition as key elements, while still focusing heavily on report generation features that are secondary in a modern DBMS. On the other hand, research projects with far greater scope, incorporating artificial intelligence and natural language interfaces, held out the possibility that data management systems would be subsumed by software with much broader capabilities in the near future.

The current consensus on the functionality of a DBMS arose both from industry trends (e.g., increasing specialization in the independent software industry) and standardization efforts. The Conference on Data Systems Language (CODASYL) was a consortium originally formed to develop a "business language", but which later worked on a variety of computing standards issues, including the development of the business programming language COBOL. In 1965, CODASYL formed a committee later known as the Data Base Task Group (DBTG), which published the first standards for database management systems (Olle 1978). CODASYL arrived at a fairly general definition of the role of data description and data manipulation languages, while proposing concrete interfaces. In general outlines, it resembles that of modern systems.

The deployment of an actual interoperable standard was still quite far off in the marketplace. The functionality proposed by CODASYL was beyond that offered by most database products. Furthermore the underlying model of CODASYL was not that of today's systems. Instead they were based on the *network database model*, an extension of the hierarchic database model used in IBM's popular IMS software. In a network (graph) database, the possible relationships between data entities are fixed as part of the schema – for example, an automobile equipment database might consist of a mathematical graph (see Chap. 7, Appendix) of equipment types, having graph connections (edges) or relationships pointing (for instance) from cars to engines, from engines to cylinders, possibly cycling back to cars. Data is described by graphs (networks) giving the basic kinds of information items and their connections. The basic mode of querying was by navigating these relationships. A query on a database of automobile parts might start by navigating to a particular kind of engine, then moving to an engine component, and then down to a subcomponent. This does represent some abstraction – the description of the data does not deal with particular data formats, or the way that data is represented on disk. But the *navigational* paradigm behind the network model forced the data designer to anticipate all possible relationships in advance. Although the model does not specify how to choose these relationships, many of them would be based on performance considerations – in which direction a querier would be most likely to navigate. Thus the distinction between a data description and an implementation or *index* was muddied. Furthermore the navigational query language forced the query-writer to think somewhat procedurally.

In the late 1960s the *relational model* evolved as a proposed mathematical basis for database management systems. It began with articles advocating the use of mathematical set theory as the core of database query languages. Although there were many forerunners, such as the proposals of David Childs (1968), the model was crystallized in the works of Edgar F. Codd. In his seminal paper overviewing the relational model (Codd 1970), Codd summarized the state of existing approaches to modeling data as follows:

> The provision of data description tables in recently developed information systems represents a major advance toward the goal of data independence. Such tables facilitate changing certain characteristics of the data representation stored in a data bank. However, the variety of data representation characteristics which can be changed without logically

> impairing some application programs is still quite limited. Further, the model of data with which users interact is still cluttered with representational properties, particularly in regard to the representation of collections of data (as opposed to individual items).

Codd defined a simple and elegant underlying model – a definition of what a database is in mathematical terms. He went on to propose that a query language should be employed that allowed users to define any *logical* relationship or access path.

> The universality of the data sublanguage lies in its descriptive ability (not its computing ability). In a large data bank each subset of the data has a very large number of possible (and sensible) descriptions, even when we assume (as we do) that there is only a finite set of function subroutines to which the system has access for use in qualifying data for retrieval. Thus, the class of qualification expressions which can be used in a set specification must have the descriptive power of the class of well-formed formulas of an applied predicate calculus.

Codd proposed two "pure mathematical" query languages – the *relational algebr*a and the *relational calculus* – proving that they had the same expressiveness, and arguing that they could be used as a benchmark by which to judge more realistic query languages. We discuss these languages in the next section.

Codd's work had an enormous impact on database research. It focused attention on the analysis of the behavior of logical formulas on finite structures: evaluation, equivalence, and simplification of logical formulas became a fundamental part of database research. The impact on the database industry was just as large but not as immediate. The presentations of the relational model were written in a highly mathematical style. They were considered unrealistic by database vendors. Thus while the relational *model* was proposed in the late 1960s, relational databases evolved only gradually throughout the 1970s within the research community. A major breakthrough was System R (Chamberlin et al. 1981), developed at IBM research in San Jose. The project was initiated in 1974 and continued throughout the 1970s, with the first customer installation in 1977. The basic features of the relational paradigm – which we describe next – were present in System R, including transaction support, join optimization algorithms, and B-tree indexes. The major database systems of the 1980s, from IBM DB2 to Oracle, descend directly from System R. In the process of creating System R, Donald Chamberlin and Raymond Boyce proposed the SEQUEL language (Chamberlin and Boyce 1974). This evolved into the standard relational query language SQL, the first version of which was standardized by ANSI in 1986.

As the computer industry moved from mainframes and dumb terminals to networks of personal computers, database management systems migrated to the use of a client–server model. In the 1990s with the rise of the Web (see Chap. 7), database systems became an integral part of e-commerce solutions. A database server would typically sit behind a Web server; remote client requests coming via HTTP to the Web server would invoke SQL requests to the DBMS, with results sent back to the client in HTML. By 1999, the relational database market (including OO extensions), was estimated to have revenues of 11.1 billion dollars (an estimate of the market research firm IDC, quoted from Leavitt 2000).

Students

id	*first*	*last*	*credits*
1	John	Smith	123
2	Jane	Doe	44
3	Bill	Wright	67
4	Janet	Who	2

Enrollment

id	*course*	*credits*
1	automata	20
1	databases	40
2	databases	30
3	algorithms	20

Dependencies

course	*dependson*
algorithms	calculus
databases	algorithms
automata	algorithms
web	databases

Fig. 10.1 Example database for the university enrollment setting

We will now review the basic features of the *relational paradigm* – our name for the features of a "traditional relational database", representing both the core of most commercial systems and the view of databases given in most database textbooks.

A Tour of the Relational Paradigm

The relational paradigm is based on several key principles:

- Data abstraction and data definition languages
- Declarative queries
- Indexed data structures
- Algebras as compilation targets
- Cost-based optimization
- Transactions

We will go through the principles individually, using the sample of a university enrollment database given in Fig. 10.1 below.

Abstraction

Abstraction is a key principle in every area of computer science – shielding people or programs that make use of a particular software artifact from knowing details that are "internal". In the database context, this means that users of database systems should be shielded from the internals of database management – what data structures or algorithms are utilized to make access more efficient. Thus a user should be able to define only the *structure* of the data, without any information about concrete physical storage. The interface which describes the structure is a *data definition language*. Access to the data should only refer to the structure given in the definition.

While standard programming languages provide a rich variety of data structures that can be defined by a user, relational languages require the user to describe data in terms of a very simple *table* data structure: a collection of attributes, each having values in some scalar datatype. The attributes of a table are unordered, allowing the data to be returned with any ordering of columns in addition to any ordering of rows.

In the university example, the database creator would declare that there is a table for students, listing their names, their student identifier, and their number of credits. Such a *Students* table with example data is given in Fig. 10.1, along with other tables in the university enrollment database.

The relational database standard language SQL (abbreviating "Structured Query Language", although it contains sublanguages for almost every database task) provides a data definition language in the form of a repertoire of `CREATE TABLE` commands, which allows the user to describe a table, its attributes, their datatypes, along with additional internal information. For example, to create the *Students* table, one could use the following SQL command:

```
CREATE TABLE Students (
  id INT PRIMARY KEY, first TEXT, last TEXT, credits INT)
```

Relational data definition languages allow database designers to describe in a declarative format important aspects of the *semantics* of the data in a way that can be exploited by a DBMS. In particular, they can include *integrity constraints*, which give properties the data needs to satisfy in order to be considered "sane". The `PRIMARY KEY` declaration above states that the *Students* table cannot contain two rows with the same *id* field and is an example of an integrity constraint.

Relational query languages allow the user to extract information according to the structure that has been defined. One could issue a query asking for all students who have taken at least 50 credits of courses. In accordance with the data abstraction principle, queries cannot be issued based on the internals of data storage; a user could not ask for all data on a particular disk, or all data located near a particular item within storage or accessible within a particular data structure.

Declarative, Computationally Limited, Languages

The fact that data is to be retrieved via an abstract descriptive interface leaves open the question of what kind of programming infrastructure could be used to actually access it. One approach would be to use "data-item-at-a-time" programming interfaces, which allow a programmer to navigate through the database in the same way they navigate through any data structure in a general-purpose programming language (Chap. 4): issue a command to get to the entry point of the structure, say an array, and then iterate through it. If we wanted to get the students who have taken at least 50 credits of courses, such an API requires a program (in the host programming language, e.g., C, Java, C++) that issues an API command to connect to the student table, another command to access the first (in arbitrary order) row (or *tuple*) in the table and put it into a host-language variable, and then a loop in the host programming language that performs the following action: checking that the current tuple satisfies the criterion (at least 50 credits) and if so, adding it to the output, then proceeding to the next tuple via calling an API command.

Such an iterative interface is simple for a programmer to use, since it requires only the knowledge of a few basic commands – e.g., a command to get the next tuple and store the result in a local variable. It allows the programmer to exploit all the features of the host language.

There are two main drawbacks to a data-item-at-a-time approach. First of all, it does not give a way for non-programmers to access the data. Anyone who wants to get the students with at least 50 credits has to know how to program. Secondly, and perhaps more importantly, performance will suffer in this approach. The program will have to access every student record in order to access the ones of interest. Further, records will be fetched one at a time, even though the architecture of any computer would allow hundreds if not thousands of records to be transferred between disk and main memory in a single-command.

The alternative pursued in relational database systems is the use of *query languages*; access to the database is by issuing statements that define properties of the set of tuples to be retrieved, giving no indication how they should be obtained. In the example above, a user would state that they would like all student ids for students with number of credit attributes above 50. The SQL representation of this is basically a formal structured version of the natural language phrasing, given in the query Q_0 below:

```
SELECT s.id FROM Students s WHERE s.credits > 50
```

`SELECT` describes a subset of the tuples satisfying the `WHERE` ("such that") clause. The above is a very simple example, but query languages can express fairly complex subset requests. For example, a query asking for the names of students with number of credit attributes above 50 who are enrolled in databases would be the following query Q_1:

```
SELECT s.last FROM Students s
WHERE s.credits > 50 AND s.id IN (
   SELECT e.id FROM Enrollment e WHERE e.course='databases')
```

Here `IN` specifies the set membership relation.

The declarative style of set-theoretic subset interfaces allows additional abstraction: the database manager is now free to choose a procedural implementation of the set theory operations that fits the current storage structure of the data. The second approach has become the ideal for database access and also for database update and transformation – programs or users describe the collection of data items that they would like to see or to change, and the details of how to do this are left to the database manager.

There are many possible declarative languages. Prolog, for example is a paradigmatic declarative language, with no explicit control structures. It nevertheless allows one to express any possible computation, including arithmetic and recursive definitions. Relational database systems, in contrast, looked for languages with limited expressiveness – ones that can only express computations that can be performed reasonably efficiently. The motivation is to prevent users from writing

queries that cannot be executed, or queries whose execution will degrade the performance of the database manager unacceptably.

What exactly does *limited expressiveness* mean? At a minimum, it means that queries cannot be expressed in the language if they require time exponential, or more generally super-polynomial, in the database cardinality. In practice, one desires performance much better than this – for large datasets one generally wants implementations that run in time less than $C_1|D|^2 + C_2$ on a database D, where the coefficients C_1, C_2 are not enormous. One coarse benchmark for a query language is given by *polynomial-time data complexity*: for every query Q there should be a polynomial P such that the execution of Q on a database D can be performed in time at most $P(|D|)$. (As usual, $|D|$ is the cardinality of D.)

The standard query language that emerged as part of the SQL standard fulfilled the polynomial time requirement. In fact, a large fragment of the language could be translated into a much more restricted language, a variant of *first-order logic*. (See Chap. 2, Appendix G.) This fragment is the one formed from nesting the basic subsetting `SELECT...FROM... WHERE` clauses of SQL, connecting multiple clauses via the quantifiers `EXISTS` and `NOT EXISTS`, or (equivalently) with the set membership constructs `IN` and `NOT IN`. We refer to this fragment as *first-order SQL* in the remainder of this chapter. For example, the following first-order SQL query Q_2 retrieves the names of students who did not take databases:

```
SELECT s.last FROM Students
WHERE s.id NOT IN (
   SELECT e.id FROM Enrollment e WHERE e.course='databases')
```

First-order SQL translates to a simple syntactic variant of first-order logic known as *relational calculus*. Every relational calculus query can be performed in polynomial time on a Turing machine, and in constant time on a parallel machine. The translation to a predicate logic is not used for compilation of the language. (As we shall see, queries are compiled into algebraic formalisms instead.) But predicate logics are often useful for reasoning about the properties of a query language, since logics are well-understood and well-studied formalisms. There are translations in the other direction as well: for example, SQL can express all Boolean queries that can be defined in first-order logic over the relation symbols. While the polynomial time requirement represents a limit on the expressiveness of query languages, the requirement of expressing all queries in a logic gives a lower bound on expressiveness, often referred to as *relational completeness*. The ideal would be to have a query language that corresponds in expressiveness exactly to a predicate logic – preferably one with a well-established set of proof rules. Then the optimization rules of the query language could be justified by the soundness of the proof system for the logic. Relational languages do not meet this ideal – SQL is much more powerful than first-order logic, and does not correspond exactly to any well-studied logic – but they approximate it.

Just because database query languages are limited in expressiveness does not mean that users are restricted in performing certain tasks. For example, a user can

still filter data based on some complex arithmetic comparison or recursive function. The idea is not that query languages would replace general-purpose programming languages. They would only be used to express requests for information that requires searching and combining data from a large dataset. Finer filtering of information would be performed within a general-purpose language.

An example of the runtime flow for our first query Q_0 might be:

```
Q0 ← SELECT s.id FROM Students s WHERE s.credits > 50;
D0 ← newDatabaseConnection();
results ← D0.execute(Q0 );
while not end of results do
    studentRecord ← results.next();
    if good(studentRecord.credits) then print(studentRecord );
endw
```

The "`execute`" operation evaluates the query Q_0 on the stored data. The result of execution may be the transferring of all of the data into memory, or just the determination of the initial record satisfying the query, with the remaining records pulled in on demand. The "`next`" operator iterates through all of the records satisfying the query. The filter "`good`" is a function on tuples written in the host language, and could use any features available in that language. A database management system thus divides up work between the host programming language and a special-purpose language.

Indexed Data Architecture

Relational database managers were designed for datasets that would be too large to fit into a computer's main-memory. At any point in time, a portion of the data would be in memory (in a buffer cache) and this portion could be accessed and navigated quickly. The remainder would be on secondary storage (e.g., disk drives). The disk-resident data can be divided up into *blocks*, a unit that can be transferred to main-memory in one atomic operation. Query processing would involve locating relevant blocks of data on disk, transferring block by block to the buffer, and then locating the required data items by navigating a block.

The main tool relational database managers use to speed query processing is the maintenance of auxiliary data structures that allow retrieval with fewer accesses. The principal example of this are tree indexes, such as B-trees and B^+-trees. In the student example, we might create an index on *id*. If the ids range among 8-digit numbers, the first level of the tree divides these numbers into some number of intervals, and similarly each of these intervals is split into subintervals in the next node.

To find the student with id 12345678, we follow a path down the tree by locating 12345678 within the collection of intervals under the root, then within the collection of subintervals, until finally arriving at the leaf node containing the block

where the student record resides. Since the internal nodes of a tree index contain only a subset of the values for one attribute (hence only a subset of the ids), they will generally be dramatically smaller than the dataset. Still for a large dataset, the bulk of the tree would reside on disk.

The trees used are *balanced*, like many tree structures used in computing: the maximum number of levels below any given node is thus fixed, and this guarantees that the access time for an *id* value will not vary from id to id. As data is modified the tree indexes must be updated, but standard tree update algorithms can be applied to make the update time a constant factor (between 1 and 2) in the number of updates.

The key thing that distinguishes tree indexes from other tree data structures is their *branching*. Instead of using binary branching, as search trees elsewhere in computing do, the branching in B-trees is chosen so that one internal node can be stored in a single block of memory, and thus a single navigation step in a tree requires at most one data transfer step from disk to memory. Depending on the block size of the machine and the size of a data item, there might be hundreds or even thousands of entries at a given level.

Algebraic Query Plans

One of the key advantages of declarative languages is that the optimizer can choose the best implementation, using a more global view of the query than the compiler for a data-item-at-a-time language would possess.

A way to capture this extra dimension of flexibility between queries and evaluation mechanisms is via the notion of a *query plan*. A *plan* is a description of high-level steps that implement the query. Many plans can correspond to the query, and have different performance characteristics.

Consider again the query Q_1 above. A naïve query plan would correspond to the following step: getting all ids of students taking the course "databases", using an index I on the table, reading the blocks of this result one at a time; finding all the student records that correspond to these ids; scanning through them to check the number of credits, returning only those above the credit threshold; scanning through the list of student records that survive the filtering process and returning all the name fields. This plan might be represented internally within a database manager by the following expression:

$$\pi_{last}(\sigma^{scan}_{credits>50}(Students \bowtie \pi_{id}(\sigma^{I}_{course="databases"}(Enrollment)))).$$

Here, $\sigma^{I}_{course="databases"}$ refers to a *selection* operator, which uses index I to retrieve all records on a particular course; $\bowtie$ is a *join* operator, which takes two tables and merges all matching records – in this case, a table of ids and a table of student records; $\sigma^{scan}_{credits>50}$ is an operator that selects students within a student table above 50 credits, via just iterating through the table block by block; finally, π_{last} and

π_{id} refer to *projection* operators, which remove all columns from a table except, respectively, the columns *last* and *id*, eliminating duplicate rows.

Of course, many details are omitted from this plan; in particular, there are many ways of implementing the join operator $\bowtie$. This plan follows the structure of the query, and thus represents a fairly naïve implementation. The key point is that there are many other plans that implement the same query, some of which will not follow the structure of the original query closely. For example, another plan is as follows: use the index on the *Enrollment* table to get the list of records for "database", then use another index J on the *Students* table to get all student records for students having at least 50 credits; then join the two tables; finally, remove all but the *last* field, eliminating duplicates. This plan might be represented as follows:

$$\pi_{last}(\sigma^{J}_{credits>50}(Students) \bowtie \sigma^{I}_{course=''\text{databases}''}(Enrollment)).$$

The plan expressions that we are displaying are in a language called the *relational algebra*. The *relational algebra* is still declarative. It has the same advantage over formalisms such as the relational calculus as compilation formalisms for general-purpose programming languages have over the corresponding source languages: they are easier to optimize because there are fewer syntactic operators. In particular, the language is *variable-free* – there are no explicit variables in the syntax – and thus the conditions under which a new query can be formed from composing a new operator are simpler.

The process of getting from a query to an efficient plan expression consists of a translation to algebra and then transforming via applying equivalences – analogous to the application of algebraic rules such as commutativity and associativity in algebra. The standard example of such a rule is *pushing selections* inside projections or joins: a query plan

$$\sigma_{course=''\text{databases}''}(\sigma_{credits>50}(Students) \bowtie Enrollment)$$

would be converted to

$$\sigma_{credits>50}(Students) \bowtie \sigma_{course=''databases''}(Enrollment).$$

In searching through plans by applying transformations, we are exploring the space of possible implementations of the query.

Cost Estimation and Search

The translation to algebra and the use of transformation rules allows one to explore the implementation space. But two issues remain: how does one determine how efficient an implementation is? And given that the search space is large – indeed,

the collection of equivalent expressions is infinite – how does one search through it in a way that makes it likely to find the best plan?

Relational database managers approach the first question by defining heuristic cost estimates on a per-operator basis. Of course the real cost of basic operations, such as retrieving all elements that satisfy a given selection criterion, depends on the data – one cannot know it exactly without executing the query. Statistical information about the data, refreshed periodically, can provide a substitute for exact information. For example, if one stores a histogram telling what percentage of the students have credit totals in any interval of length 5 between 0 and 150, then one can get a very accurate estimate of the number of students having above 50 credits. This will allow one to estimate the cost of the selection on I in both plans above. Cost estimation of basic operations on relations is highly tied to the index structures and physical storage – it takes into account index structures, when they are present, cached data, and locality of data on disk. Thus much of the implementation of relational structures is encapsulated within a cost function, which serves as an interface to the query optimizer.

In terms of the second question, relational database managers have no universal solution for searching the space of query plans. They apply some standard search techniques, but customized to the database setting. In particular, they rely heavily on divide-and-conquer, breaking up the algebraic expression into subparts and optimizing them separately; the variable-free nature of relational algebra expressions makes it easy to analyze components of queries in the same framework as queries, which assists in defining algorithms via recursion on query structure.

ACID Transactions

Above we have focused on querying databases, but databases are also being updated concurrently with query accesses. The concurrency of updates and queries introduces many issues. Consider two users of our university database. User A is doing an update that is removing a student S from the *Students* table along with all of the student's records in the *Enrollment* table. Concurrently user B is querying for the average number of courses for any student, a query that involves both the number of total courses in the *Enrollment* table and the number of students. The high-level query of B translates to a number of access operations on the database, while the update performed by A translates to changing records in two distinct tables. If the low-level operations are interleaved in an arbitrary order at the database, then the users may see anomalous results: the average seen by B might reflect a student table that includes student S, but an *Enrollment* table that lacks the records of S, or vice versa. Indeed, the average seen by B might reflect a table including only a portion of the enrollment records of S.

The issue is related to the level of abstraction provided to users by a database system; a complex data-intensive activity like querying for an average is provided as a single primitive to user B, who will consider it to be *atomic* – something that cannot properly overlap with other database activities. At the very least, the

database should support this. Furthermore, user A would like an even higher-level of abstraction: A would like the two updates together to be considered as a single primitive, which should not properly overlap with other database activities. A DBMS provides additional language support that allows A to do this – to state that the delete of student S and the elimination of S's enrollment records represents a *transaction*, which should have the properties of a single indivisible action.

Above we have spoken of an action or sequence of actions being treated as "atomic" or "indivisible". But what does this mean in practice? Relational databases have formalized this by the requirement that transactions satisfy the following properties:

Atomicity
No transaction should be "implemented in part". If there is a failure in the process of performing one of the updates in the transaction (e.g., due to hardware failure or integrity constraint failure), then any other updates that have been applied should be *rolled back*, and their effects should not be seen by other database users.

Consistency
Transactions should leave the database in a consistent state: one in which all integrity constraints hold.

Isolation
Until a transaction has completed, no concurrent user should see results that are impacted by the updates in the transaction.

Durability
Conversely, once a transaction has completed, its results should not be rolled back regardless of hardware or software failures. We say that the transaction should be durable.

Support for transactions that satisfy the properties above – abbreviated as *ACID transactions* – is one of the main goals of relational systems. ACIDity can certainly be achieved by running each transaction serially: assigning each one a timestamp, and running the transactions in timestamp order (queuing those with lower timestamp), rolling back the transactions that do not complete. Logging mechanisms can be used to track the impact of the transaction to enable rollback. The problem is that this can lead to unacceptable delays in running updates and queries. The goal is then not just to enforce the ACID properties, but to enforce them while allowing updates and queries to proceed without blocking whenever this would not destroy ACIDity – that is, to allow as much concurrency as possible.

Locking mechanisms are the most popular technique for managing concurrent use of the data; transactions are only allowed to modify data items that are not locked by another transaction, and when they act on an item they receive a lock on it, which generally remains in place until the transaction completes or aborts. When a transaction queries data it receives a weaker lock on the data, one which

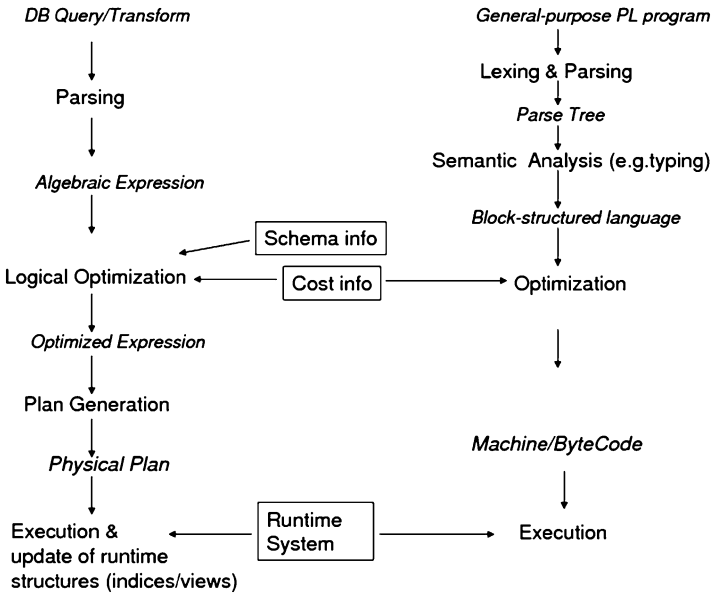

Fig.10.2 Processes for database programs and general-purpose programming languages

allows other transactions to query the same data but not to update it. Lock-based concurrency control has many variants, particularly concerning the granularity at which data is locked.

The Evaluation Pipeline of the Relational Paradigm

The relational paradigm gives a flow of processing for queries that is closely-modeled on the flow of processing of programs in a general-purpose programming language (GPPL), such as C or Java. Figure 10.2 shows a comparison of the processing flow at query/program evaluation time for a general-purpose program and for a database query or transform. In both cases, source language expressions are parsed, and eventually arise at a form more suitable for optimization.

The algebraic expressions are run through a *logical optimizer* which has an abstract interface to information about program executions. The optimized expression is either translated directly into an executable plan, or is run with the help of a runtime system. In the case of a DBMS, the runtime system would include indexes and other auxiliary data structures.

Several differences stand out between the two scenarios:

- Database languages are much smaller and syntactically less complex than GPPLs, and hence the parsing stage is fairly uninteresting. Similarly, the semantic analysis phase is often much simpler than for a GPPL.

- Database queries are often issued interactively, or sent from across a network. Thus optimization as a rule must usually be quite a small phase – in seconds if not milliseconds. In the case of GPPLs, the common case is that the program text is available for some time before execution, making a more robust optimization phase possible.
- GPPLs often make use of interesting runtime data-structures – byte-code interpreters, or garbage-collectors. But not all GPPLs do, and one generally cannot say that a particular program requires auxiliary runtime structures. In most applications database queries could not be executed at all without the help of large and complex runtime structures. These structures, in turn, need to be maintained after program execution (see bottom left in the figure), which may be as complex and as time-consuming as execution itself. Designing and maintaining these structures thus plays a central role in DBR.
  Related to the above, DBMS systems are often tuned for many runs of a query or set of queries, not just for an individual execution. The initial population and period maintenance of runtime structures may thus represent an independent process, taking place far prior to execution or at intervals between executions.
- A DBMS makes use of not only a program text, but information about the semantics of the data – the schema. There is no corresponding standard input description for GPPLs.
- A distinction that is not exhibited in the figure, but has an even greater impact, is that, typically, database programs run in a heavily concurrent environment, with hundreds, even tens of thousands of concurrent queries and updates sharing the same data. Hence the handling of concurrency is paramount.
- Both GPPLs and DBMSs can be used in a distributed setting where resources (data, computing power, etc.) are distributed over a computer network. However, for reasons that we will explain further on, even when there is no inherent distribution in these resources, it is common for a DBMS to manage its data in a distributed manner.

Let us return to the question of the manifold nature of database research mentioned in the introduction. We said that part of database research is driven by improving the performance of existing database management products, while other DBR is geared towards exploring the possibilities for managing data and extending the use of database management techniques.

If we consider the first kind of research, parallels with programming languages suggest that its structure could be broken down along the same lines as for PL research – there is research on optimization, research on improving runtime data structures, concurrency, etc. Of course, as the figure shows, DBR in no sense reduces to PL research, and indeed the research on DB performance has not been closely-tied to work in PL. Still in *rough* analogy with programming language work, one would expect the structure of DBR to follow the lines of the flow on the left-hand side of Fig. 10.2. And indeed much of the first kind of database research naturally works in exactly this way. We will refer to this as *core database research*,

and we will give an idea of some aspects of it in the next section, following the processing flow on the left of Fig. 10.2 in our tour.

For research extending the functionality of DBMSs, and considering ways in which data *could* be managed, there is a dichotomy: some of the work tries to preserve the flow of processing in the relational paradigm, but with some new functionality at the query language level. Some of this work takes as a given the relational perspective of database query languages as a logical formalism, and looks at to what extent other logics could be evaluated in the same way as first-order predicate logic on relations. A second line in the more speculative kind of DBR looks at more radical changes in the processing pipeline, which have no analog in programming languages. We will overview both of these extensions further in this chapter.

Core Database Research Sampler

In this section, we provide a sampler of what research in the database field has focused on and accomplished over the years concerning the processing pipeline of relational database management systems, as was presented in the previous section. Following Fig. 10.2, we start at the query level, then move to logical optimizations, physical optimizations, down to the execution of the query on actual hardware. Each research area is represented by a selection of significant research results, with no intention of exhaustiveness or objectivity in the choice. We also indicate the impact research has had on the design of modern DBMSs, discussing whether models, algorithms, and data structures from the scientific literature have been implemented in widely used systems.

Query Languages

We previously explained how first-order SQL has nice logical and algebraic interpretations in terms of relational calculus and relational algebra. However, even the very first version of the SQL query language (Chamberlin and Boyce 1974) went beyond that fragment and included *aggregate functions* and *grouping*. Over the years, the standard and implementations of the SQL query language evolved towards more and more expressive power. We present in this section the additions that have been made to SQL to overcome some of its limitations, both in the standard and in actual implementations, that do not always follow it strictly.

Aggregation

One of the most common functions of database management systems is to compute summaries of existing data by aggregating the numerical values that appear in these

tables. Going back to the example database of Fig. 10.1, one can for instance ask the following questions:

- What is the average number of credits obtained by students?
- How many students have more than 50 credits?
- What is the maximum number of credits earned by a student enrolled in the databases course?
- For each course, what is the average number of credits of students enrolled in this course?

All these queries make use of an *aggregation function* to compute the average, count, or maximum of a collection of values. The last query also uses a *grouping operator*, where the aggregation is performed for each group of results to a subquery that have the same value for a given attribute. As already mentioned, both aggregate functions and grouping are basic features of SQL. They correspond to forming families of sets.The last query can for instance be expressed as:

```
SELECT e.course, AVG(s.credits)
FROM Students s, Enrollment e
WHERE s.id = e.id
GROUP BY e.course
```

It is also possible to define extensions of the relational algebra for queries that involve aggregation and grouping (Klug 1982; Libkin 2003). Using a similar notation as in Libkin (2004), a relational algebra expression for the query above is:

$$Group_{course}[\lambda S.\mathrm{Avg}(S)](\pi_{course,credits}(Students \bowtie Enrollment)).$$

As in the non-aggregate case, database management systems use such algebraic expressions to represent and manipulate query plans.

In essence, the queries definable in the early versions of SQL (Chamberlin and Boyce 1974; ISO 1987) are the ones of this aggregation and grouping algebra. Queries with aggregation and grouping, with their standard syntax and semantics, are supported by all relational database management systems. In the following, we will refer to this language as *vanilla SQL* to distinguish it from more recent and less well-supported additions.

Recursion

A natural question is that of the expressive power of vanilla SQL. Can all "reasonably simple" queries that one may want to ask over a relational database be expressed in SQL? Again, consider the university enrollment example. The table *Dependencies* lists the courses that a student must have followed in the past in order to get enrolled in a given course. Suppose that a new student aims at taking the Web course. Then he needs to query the database to retrieve all courses this one depends

on, so as to plan his curriculum. This is a simple enough request: if the table *Dependencies* is seen as the relation defining a directed graph, the problem becomes determining all nodes in the graph reachable from the "web" node. This can be solved in time linear in the size of the relation by simple depth-first or breadth-first graph search algorithms (in other words, by computing the transitive closure of the relation); because of the inherent recursion in these algorithms, such a query is called *recursive*. We have isolated earlier in the chapter polynomial-time data complexity as an indicator of the limits of expressiveness for database query language. The course dependency query and other similar recursive queries fit this criterion. Are they expressible in vanilla SQL?

It is well established that recursion cannot be expressed in first-order logic (and thus, in the relational calculus). This can be proved using a *locality* (Libkin 2004) argument: a relational algebra expression is unable to distinguish between two nodes in a graph whose neighborhood of a certain radius are isomorphic, while computing the transitive closure is essentially a non-local operation. It turns out that the same result holds for vanilla SQL (Libkin 2003), with aggregation and grouping, when datatypes are unordered (the problem is more complex and still open for ordered datatypes).

The (apparent) inability to write simple recursive queries in SQL has led the designers of the SQL3 standard (ISO 1999) to add to the language the `WITH RECURSIVE` feature that enables recursion. As an example, the course dependency query can be written as:

```
WITH RECURSIVE Closure(course) AS (
  VALUES('web')
  UNION
  SELECT d.dependson FROM Dependencies d, Closure c
  WHERE d.course = c.course
) SELECT * FROM Closure
```

Support for this kind of query in DBMSs varies. In Oracle, for instance, it is not possible to use `WITH RECURSIVE` queries at the time of this writing. A similar feature, however, has been available in Oracle since the early 1980s (Stocker et al. 1984) with the proprietary `CONNECT BY` operator, which demonstrates the early interest in such a functionality. IBM DB2, Microsoft SQL Server, and PostgreSQL all support `WITH RECURSIVE`, while other less feature-rich DBMSs such as MySQL do not allow any form of recursive queries, short of stored procedures.

Stored Procedures

The components of the SQL query language mentioned so far (the relational algebra, aggregation and grouping, recursion) all have in common a polynomial-time data complexity. As already discussed, this is a design choice, to avoid queries that would be too costly to evaluate. The philosophy was that more complex

processing of the data would be done outside the database management systems, in applications written in traditional programming languages. This may mean, however, redundant implementation of data-related functionalities (e.g., data validation) in all applications (possibly written in different programming languages) that interact with a given database. For this reason, users have felt the need to move larger parts of the application logic, in the form of arbitrary code that manipulates data, into the database management system. Database vendors have thus offered the possibility to implement *stored procedures* and *user-defined functions* directly in the DBMS, using extensions of the SQL language with control flow statements (variable assignment, tests, loops, etc.). These procedures are stored in the database itself, along with the data. This has led in turn to the addition of stored procedures to the SQL standard, under the name SQL/PSM (ISO 1996) (for *persistent stored modules*). Though few vendors follow this standard to the letter and there are many variations in the actual stored procedure languages used in DBMSs, all major systems provide this functionality. Oracle's stored procedure language, PL/SQL, introduced in 1992, has been especially influential.

Using stored procedures, it is possible to implement arbitrary processing of the data inside the database management system (in other terms, the addition of stored procedures make the SQL language Turing-complete). As a consequence, queries making use of stored procedures do not have any guarantee of polynomial-time data complexity and most query optimization techniques are not applicable any more. The database management system focuses on optimizing subparts of the stored procedures that do not make use of control flow statements.

The relational algebra, aggregation and grouping, recursive queries stored procedures and user-defined functions are the tools that modern database management systems provide to query relational databases. Other commonly available features of the query language either add syntactic sugar on top of these basic functionalities, or allow the querying of other kinds of data structures, such as XML documents (see further), geospatial coordinates, or plain text queried through keyword search.

Logical Optimizations

Let us move to the realm of query optimization. The goal here is to find an efficient way of evaluating a user's query. As already mentioned, database management systems do this in two ways: by first rewriting the query into a form that is easier to evaluate, independently of the data it runs on, and then by generating a set of possible evaluation plans for the query and using statistical information to choose an efficient one for this particular database. We are looking now at the former type of techniques, that we call *logical optimizations*. Plan generation and statistics-based cost estimation are described further. Logical optimizations can either be *local* (the query is rewritten parts by parts) or *global* (an optimal rewriting of the query as a whole is sought for). Since optimizations considered here are

independent of the actual data they are particularly useful when the same query is run multiple times over different database instances.

Local Optimizations

For reasons that we shall attempt to explain further on, though research has considered and proposed both local and global logical optimization strategies, DBMSs mostly use local optimizations. To go beyond what is presented in this section, a good starting point is Chaudhuri (1998).

Equivalence Rules

The first idea used for query rewriting has already been mentioned: exploiting equivalence rules of relational algebra expressions (especially, commutativity or distributivity of operators). Thus, it is often more efficient to push selections inside joins, i.e., evaluate the selection operator in each relations before joining two relations, or to distribute projection over union, i.e. to transform $\pi_A(R_1 \cup R_2)$ into $\pi_A(R_1) \cup \pi_A(R_2)$. It is not always clear, however, when applying a given equivalence rule makes the rewritten query more efficient. Therefore equivalence rules are also used extensively for generating the space of query plans a cost-based estimator chooses from. Equivalence rules of relational algebra expressions involving classical operators are folklore, but each time a new operator has been considered, new equivalence rules have been investigated. This is the case, for instance, in Rosenthal and Galindo-Legaria (1990) with the *outer join* operator that retains every tuple of one of the two tables being joined, even if no matching tuple exists in the other table. Equivalence rules are an important component of the query optimizer of all DBMSs.

Unnesting Complex Queries

Another form of logical optimization at a local level deals with *nested* SQL queries. SQL offers the possibility of expressing complex queries that use nested sub-queries, especially in the `WHERE` clause:

```
SELECT s.first, s.last
FROM Students s
WHERE s.id IN (
   SELECT e.id FROM Enrollment e WHERE e.course='databases')
```

A naïve evaluation of this query, which asks for names of students enrolled in the "databases" course, would enumerate all tuples of the *Students* table, and, for each of them, would evaluate the sub-query and return the tuple if the sub-query returns the identifier of the student. Obviously, in this particular example, such complex processing is not required, since the query and its sub-query are *uncorrelated*:

the sub-query does not refer to the current tuple of the main query. This means the sub-query can be evaluated once and its results used for matching identifiers from the *Students* table.

There are more complex examples of nested queries, for which such a simple strategy cannot work. Consider for instance the following query, that retrieves names of students enrolled in a course that would allow them to graduate (assuming they need 150 credits to graduate, counting those they were already awarded):

```
SELECT s.first, s.last
FROM Students s
WHERE s.credits < 150 AND s.id IN (
  SELECT e.id FROM Enrollment e
  WHERE e.credits > 150 − s.credits)
```

Here, the main query and its sub-query are *correlated*: the sub-query has a condition on the value of the current tuple of the *Students* table.

Research has investigated the conditions under which nested queries could be unnested and rewritten as simple one-block queries. The seminal work (Kim 1982) proposes an algorithm to simplify nested queries, which depends of the kind of correlations existing between a query and its sub-query. In this example, a simplification is possible, and yields:

```
SELECT s.last
FROM Students s, Enrollment e
WHERE s.id = e.id AND s.credits < 150
  AND e.credits > 150 − s.credits
```

This rewritten query can be shown to be equivalent to the original one. This *decorrelation* procedure is actually very simple when the two tables are joined by the `IN` operator and when the correlation between the two tables does not involve any aggregate function: just put all conditions of the sub-query in the main `WHERE` clause, and replace the `IN` operator with an equality join. Other works, such as Dayal (1987), have extended the decorrelation procedure of Kim (1982) to support, for instance, grouping, and other forms of correlation between a query and its sub-query.

Obviously, whenever such simplifications are possible (which is not always the case!) they can be applied repeatedly to complex sub-queries to reduce the number of nested blocks, possibly reducing them to a simple `SELECT-FROM-WHERE` query. The reduced query is typically more efficient to directly evaluate, leads to a search space of query plans of reduced size, and is also more easily subject to other forms of logical optimizations, such as static analysis. For these reasons, modern-day query optimizers perform such unnesting, at least in simple cases (Lorentz 2010).

Global Optimizations: Static Analysis

We discuss here a global approach to query optimization, based on *static analysis*: independently of actual data, determine important characteristics of a query as a

whole that can be used, in particular, to determine whether the query can actually return any result, what is the optimal evaluation order, or how to rewrite it into a simpler one.

We limit ourselves in this section to the well-understood case of *conjunctive queries*, the fragment of the relational calculus without disjunction, negation, or universal quantifiers. A conjunctive query is thus a conjunction of relational facts (also called *subgoals*) involving either output variables, or existentially quantified variables, or constants, for instance the query Q_1, previously introduced:

$$Q_1(last) := \exists i \exists f \exists c. Students(i, f, last, c) \wedge c > 50 \wedge Enrollment(i, "databases").$$

Equivalently, this can be seen as a fragment or the relational algebra with the selection, projection, join, and cross product operators, or also as simple SQL queries involving only the `SELECT`, `FROM`, and `WHERE` keywords.

The most basic static analysis problem is that of *satisfiability*: does there exist a database for which the query returns a non-empty result? In the case of conjunctive queries, and without any restrictions on the data, it is easy to see the answer is always yes, because of the monotonicity of the query language. For the full relational calculus, satisfiability is undecidable. The undecidability of the existence of arbitrary models of a first-order logic formula is a consequence of Gödel's incompleteness theorem (see Chap. 2, Appendix G); however, the proof of the undecidability of relational calculus satisfiability (Di Paola 1969) relies on the undecidability of the existence of *finite* models of a first-order logic formula, a result known as Trakhtenbrot's theorem (Trakhtenbrot 1963).

Query Evaluation and Query Containment

Another fundamental problem in static analysis is *query containment*: query Q_1 is said to be *contained* in query Q_2 ($Q_1 \subseteq Q_2$) if, for all databases D, the set of results of Q_1 over D, $Q_1(D)$, is a subset of $Q_2(D)$. For conjunctive queries, a fundamental result known as the *homomorphism theorem* (Chandra and Merlin 1977) relates query containment and query evaluation through the *canonical database* D^Q of a conjunctive query, constructed as follows: each subgoal occurring in the query forms one tuple of its canonical database, and each constant occurring in the query or output variable of the query is the sole tuple of a new unary relation. A *homomorphism* between two relational databases is a mapping h from one to the other such that if $R(c_1 \dots c_n)$ is a tuple of the first database, then $R(h(c_1) \dots h(c_n))$ is a tuple of the second. The homomorphism theorem states that the following three statements are equivalent:

1. $Q_1 \subseteq Q_2$;
2. The output variables of Q_1 are in $Q_2(D_1^Q)$;
3. There is a homomorphism from D_2^Q to D_1^Q.

In other words, testing containment amounts to evaluating a query, and, conversely, evaluating a query Q over D amounts to testing containment of a query for

which D is the canonical model in Q. An efficient algorithm for query containment would thus give us an efficient query evaluation strategy. It is easy to see, however, that conjunctive query containment is an NP-complete problem (Chandra and Merlin 1977), and (thus) that in terms of *combined complexity* (Vardi 1985) (i.e., the complexity when both the data and the query are part of the input), query evaluation is NP-complete. In fact, satisfiability of propositional formulas in conjunctive normal form (Chap. 2, Appendix 1), the most well-known NP-complete problem, (see Chap. 13), is a special case of query evaluation over a database where the domain of each relation has only two elements.

Acyclic Queries

Given this intractability of conjunctive query evaluation in the query size, research on query optimization has investigated subclasses of conjunctive queries for which (a) containment is in polynomial time and evaluation is therefore in polynomial time in combined complexity and (b) the recognition problem – determining whether a query belongs to the subclass – is tractable. In particular, the class of *acyclic* queries (Chekuri and Rajaraman 2000) has been widely studied.

To define acyclicity, first consider a simple case when all relations used in a conjunctive query have arity 1 or 2. We define the graph of such a query. Nodes of the graph are constants or variables occurring in the query, and there is an edge between two nodes if there is an atom that involves both these nodes. A query is acyclic if its graph is acyclic. For example, the query $\exists x \exists y \exists z R(x, y) \wedge R(y, z)$ is acyclic, while $\exists x \exists y \exists z R(x, y) \wedge R(y, z) \wedge R(z, x)$ is cyclic. For general conjunctive queries with no restriction on the arity of the relation, the definition of acyclicity involves the hypergraph of the query, and essentially means that there exists a tree-like decomposition of the hypergraph; the precise definition is a bit technical and can be found in Beeri et al. (1981), along with equivalent characterizations.

Acyclic queries are of particular interest because query evaluation can be performed in polynomial-time combined complexity (Chekuri and Rajaraman 2000). Intuitively, for a query with no output variables, it is possible to use the tree decomposition of the query as a query plan where join operators can be replaced with *semijoins* (the semijoin of two relations is the tuples of the first relation for which a matching tuple exists in the second one). One can easily see that testing acyclicity is also tractable. Numerous queries encountered in practice are indeed acyclic; this is the case of most queries related to the university enrollment example encountered so far. But some simple queries are cyclic. Consider for instance the following one, that checks if there is any student enrolled at the same time in a course and one of its prerequisites:

$$\mathrm{Q}_3 := \exists i \exists c_1 \exists c_2 . \mathit{Enrollment}(i, c_1) \wedge \mathit{Enrollment}(i, c_2) \wedge \mathit{Dependencies}(c_1, c_2).$$

This query is obviously cyclic. In order to extend tractable query containment to simple yet cyclic queries, the notion of *treewidth* (Chekuri and Rajaraman 2000)

and the more general *hypertree-width* (Gottlob et al. 2002b) have been introduced to characterize the "degree of cyclicity" of a query; intuitively, the tree decomposition of a query is allowed to have more than one subgoal, and the maximum number of these subgoals in the tree gives the width. We omit the precise definition of these concepts, but treewidth and hypertree-width have in common that acyclic queries have width of 1, a "cycle" query such as Q_3 has width of 2, and, more generally, the more complex the sharing patterns of variables across subgoals, the larger the width. Though computing the treewidth or hypertree-width is NP-hard, there are polynomial-time algorithms that check whether a query has width less than or equal to a given constant k. Furthermore, if a query has treewidth or hypertree-width at most k for any fixed k, query containment can be tested in polynomial time, and thus query evaluation is polynomial-time in the size of the query and data. The advantage of hypertree-width over treewidth is that for some queries, the hypertree-width can be arbitrarily smaller than the treewidth.

Query Minimization

A query optimization problem orthogonal to finding an efficient evaluation strategy is *minimization*: does the query have a minimum number of subgoals among all equivalent queries, and if not, how can we rewrite it into an equivalent, minimal, query? As a rule of thumb, the shorter a conjunctive query is, the faster it can be processed. The problem of query minimization is especially significant when the query is not hand-written but automatically generated, e.g., by a content management system or in data integration contexts. Such automatically generated queries are commonly much longer than minimal equivalent queries, involving many join computations that could be avoided.

A consequence of the homomorphism theorem is that every conjunctive query has a unique minimal equivalent query (up to the renaming of variables). Furthermore this minimal query is a homomorphic image of the original query. A strategy for query minimization (Abiteboul et al. 1995) is thus to repeatedly try reducing the overall number of subgoals of the query by mapping variables to constants or other existing variables, testing at each step whether the reduced query is still equivalent to the original query. When no further reduction in the number of subgoals is possible, we have obtained the minimal query. Consider for instance the Boolean query $\exists x \exists y \exists z . R(x, y) \wedge R(z, 5)$. By mapping x to z and y to 5 we obtain the reduced query $\exists z . R(z, 5)$ which can be checked to be equivalent to the original query. Further minimization is obviously impossible, so this is the minimized query.

Note that the homomorphic image of an acyclic query is also acyclic. This means that this minimization procedure can be run in polynomial time over acyclic conjunctive queries. A query optimizer can thus start by checking whether a given query is acyclic, and if so, minimize it, to reduce the cost of its evaluation, and also evaluate it in polynomial time. Once again, this approach can be generalized to queries of bounded treewidth or hypertree-width.

Use of Static Analysis in DBMSs

Basic static analysis is used in relational database management systems, e.g., to test if a query is acyclic in order to replace joins with semijoins (Lorentz 2010). More advanced treewidth-based query evaluation and query minimization, to the best of our knowledge, are not used in any major database management systems, despite the potential usefulness of such techniques in practical applications (Kunen and Suciu 2002). The reason may be that queries are often already minimal and in an easily evaluable form when hand-written; more and more scenarios call for automatically generated queries of arbitrary structure, however. Another reason is the generally good performance of classical cost-based optimizers.

Between Logical and Physical Optimizations: Views

Before moving down the query processing pipeline to plan generation and cost-based query optimization, let us remain at the logical level to discuss *views*: a *view* is a named query that can be used in other queries as if it were a base table in the database. Views can be defined in SQL like this:

```
CREATE VIEW PredictedCredits AS
  SELECT s.id, s.credits + SUM(e.credits) AS credits
  FROM Students s, Enrollment e
  WHERE s.id = e.id
  GROUP BY e.id
```

This statement creates a view *PredictedCredits* that contains the number of credits for each active student at the end of the term, provided they pass all courses they enrolled in. This view can now be referred to in subsequent queries (e.g., `SELECT AVG(`*credits*`) FROM` *PredictedCredits*).

Views can either be *virtual* or *materialized*. Virtual views are just aliases for subqueries; when they are used inside a query, they are substituted with their definition. The full expansion of queries that use views can become relatively complicated, and the unnesting procedures discussed in the previous section can be helpful to optimize them. In materialized views, the situation is different: the query defining the view is evaluated to produce the result table, and this result table is stored and can be directly used as a table in query evaluation.

Virtual views can be used as an extra abstraction layer on top of the original data: one sometimes consider views as belonging to an *external layer* that users access, on top of the *logical layer* of relational tables that organize the data, on top of the *physical layer* of indexes and data storage structures. Separating views from data allows them to be used for confidentiality purposes: the view *PredictedCredits* can be published and used for statistics purposes, without the identity of the

students being unveiled. Virtual views are also crucial in data integration contexts, when a source does not provide access to all its data, but just to a view over its data.

Materialized views are generally used for optimization purposes: if a complicated query or sub-query is often used, a materialized view defined by this query avoids repeating the same computation again and again, functioning as a cache over the data. This can be done in two ways: either the user needs to refer to the materialized view in the query for it to be used, or the materialized view is automatically used whenever useful, the database engine rewriting the query to make use of this cached data, typically using query containment tests to check for the usability of the view.

All relational DBMSs support virtual views. Materialized views are not part of the SQL standard (there is the possibility of defining tables by a query, but such tables are not *maintained* as we discuss further) but major systems offer the possibility of creating them, with a proprietary syntax. At creation, it is generally possible to specify whether the materialized view should be used for optimization purposes when finding a rewriting of a query (e.g., the `ENABLE QUERY OPTIMIZATION` clause of DB2's materialized query tables).

Updates in Relational Databases

Most research about views in relational databases relates to their interaction with database updates. Until now, we have mostly had a static view of databases: the content of tables is fixed, and we interact with them through queries. Obviously, in most applications, tables change over time, as new data appears, data gets modified, and old data is removed from the database. These three basic operations can be carried out in SQL as follows:

```
INSERT INTO Students VALUES(5,'Alice','Liddell',0)
UPDATE Students SET credits = credits + 10 WHERE id = 3
DELETE FROM Students WHERE credits < 50
```

These three update operations respectively insert a new student in the database, increase the credits of a given student, and delete a student from the *Students* table. Note that the location of the tuple to update or delete is given in the `WHERE` clause similarly as it would be expressed in a query: the idea of using locator queries to express updates is a very general one.

Two fundamental problems arise when views are defined over dynamic data. First, materialized views need to be updated whenever the result of their defining query changes because of updates in the database; this is known as *view maintenance*. Second, since views can be used as an external layer that users can query without knowledge of the logical organization of the data, they should also be able to update the data through views, and this update should be propagated to the original data: this is the *view update* problem. We now elaborate on these two problems.

View Maintenance

The view maintenance problem is to determine how to efficiently maintain an up-to-date materialized view when its base tables are updated. Consider again the view *PredictedCredits* that we assume has been materialized. It is clear that each time a student is removed from the *Students* table, or the current credits of a student are updated, or a new tuple is inserted into the *Enrollment* table, the relevant portion of the view needs to be updated as well.

In order to avoid unneeded computations, the system needs to detect whether a view can be affected by a given update, i.e., whether the update is *relevant* to the query. For instance, no update in the *Dependencies* table, and no modification of the name of a student, can have an impact on the *PredictedCredits* view. Static analysis approaches can be used to determine the potential impact of an update on a view, independently of the current data. Once an update is found relevant to the view, the query defining the view can be evaluated again and the view reconstructed.

In most cases, it is possible to do better, with an *incremental maintenance* approach, that aims at avoiding this recalculation step together, and just incrementally maintaining the view by adding or removing individual tuples. Let us see a practical example, with the simple yet elegant *counting algorithm* (Gupta et al. 1993) for incremental view maintenance. Consider the following (materialized) view, that lists all courses at least one student is enrolled in:

```
CREATE VIEW Courses AS
  SELECT DISTINCT e.course FROM Enrollment e
```

The main idea of the algorithm is to store in the view, in addition to the tuples, an extra counter that indicates how many *derivations* of this tuple can be found in the database. For example, the "databases" course appears twice in the table *Enrollment* so there are two different derivations of the tuple ("databases") in the view *Courses*. Consider now an update on the table *Enrollment*. For simplicity, we assume it is either an insertion or deletion, modifications being dealt with as a sequence of a deletion and an insertion (it is possible to extend the algorithm to deal with modifications in a direct manner). We describe how the view is maintained. If we deal with an insertion, let c be the projection of the new tuple on the attribute *course*. The value c is searched in the materialized view *Courses*. If it occurs, the corresponding count is incremented by 1; otherwise, it is inserted, with a count of 1. For a deletion, we proceed similarly: we decrease the counter associated with the course deleted, and if it reaches 0, we delete the tuple from the view. Such a simple procedure can be defined for a large class of queries, with support for aggregation or negation. For a broader outlook on view maintenance approaches, see Gupta and Mumick (1995).

DBMSs that support materialized views, such as Oracle or DB2, allow specifying at view creation time whether a view should be maintained automatically, and, if so, whether the view should be entirely recomputed after each update operation or incrementally maintained (when possible) (Lorentz 2010).

View Update

The view update problem is the converse of the view maintenance problem. Instead of determining the consequences on a view of an update on the database, we now look at how to translate on the databases an update on a view. Imagine a secretary with access to the sole view *Courses* needs to change the name of the "automata" course to "formal languages". Does this make sense? In other words, is there any reasonable translation of this update operation on the table *Enrollment*? In this case, it seems there is: just replace every occurrence of "automata" with "formal languages" in the table *Enrollment*. What if someone with access to the *PredictedCredits* view wants to change the credits of a given student? Now, there does not seem to be any reasonable way to translate this into a database update operation, since the *credits* attribute of the view has been computed as an aggregate of several values, and it is unclear which value should be changed.

The view update problem consists in determining in which cases updating a database through a view makes sense, and when it does, what the most reasonable translation of the view update is. In Keller (1985), algorithms are proposed for view update translation when views are defined using simple conjunctive queries with all join variables exported in the view. In addition to these algorithms, this work is of particular interest because it identifies a number of criteria that a view update translation should verify to be "reasonable":

1. The translation should not have any effect on the tuples not exported in the view.
2. The translation should affect at most once a database tuple.
3. The translation should be minimal, i.e., there should not be unnecessary operations.

The SQL standard supports updatable views only when the view is defined using a single table, and no aggregation or grouping is used. Implementations may go beyond that and sometimes allow updating simple multiple-table views. The standard `WITH CHECK OPTION` clause that can be used in view definitions states that updates that would cause changes to tuples not visible in the view should be disallowed.

Plan Generation

Looking back at Fig. 10.2, we arrive at the step where query plans are generated and the physical plan that will be run on the actual data is chosen. In order to decide on a query evaluation strategy, query optimizers are built out of three components:

- Logical rewriting rules and index access strategies that are used to generate, given a query, its possible execution plans

- A cost model for estimating the cost of a plan, typically based on the expected number of disk accesses and CPU use of every atomic operation; for the estimate to be precise, it needs to be based on statistical information about the data
- A search strategy that guides the optimizer in exploring the space of possible evaluation plans

The design of the query optimizer, and in particular, the heuristics for estimating plan costs, are an important component of database management systems, that is kept as a secret in commercial DBMSs. We now present in more detail research about using histograms for storing statistical information, and how these statistics can be used to estimate the cost of a query, before presenting the architecture of a typical query optimizer. For further reading on query optimization, we refer the reader to (Chaudhuri 1998).

Cost Estimation and Histograms

Consider the query $Q_{>50} = \sigma_{credits>50}(Students)$. Such a simple query has usually at most two possible evaluation plans: either the table *Students* is linearly scanned, and all tuples with more than 50 credits are returned, or an ordered indexed on the attribute *credits* is browsed to retrieve all relevant tuples. The *Students* table is probably stored in the order of its primary key, *id*, however; this means the index of *credits* is a *secondary* index that stores, for each possible value, a list of pointers to all corresponding tuples in the database (a *primary* index on *id* could avoid this extra indirection).

Let us try to build a cost estimate of these two query plans, based on a simple cost model that only looks at the number of disk *pages* accessed. A page is the elementary unit of storage used by the DBMS; retrieving the whole content of a page is considered as an atomic operation, while accessing another page requires a costly random-access seek. Assuming a typical page size of 4 kilobytes and that 64 bytes are required to store each tuple of the *Students* table, the whole table uses $N/64$ pages where N is the number of students. Consequently, a linear scan of the table has a cost of $N \times 64/4096 = N/64$. The cost of using the index can be decomposed as follows: first, looking up 50 in the index; second, accessing all index entries for value equal or greater than 50; third, accessing all tuples pointed to by these index entries. Let C be the number of different credit values. Assuming storing a credit value requires 2 bytes, index lookup using a B^+ tree structure has a cost of $\log_{512} C$ (512 is half the number of entries one could store in a leaf node, see Silberschatz et al. 2010). The number of pages in the index entries accessed is roughly $N_{>50}/1024$ if 4 bytes are used for storing a pointer. Here $N_{>50}$ is the number of entries having value above 50. Finally, the number of pages accessed while retrieving tuples can be as large as $N_{>50}$ since two successive tuples are typically not contiguous (it would be possible to refine a little bit this estimate by considering the probability that a tuple is in the same page as a previously accessed tuple, provided that this page was cached). Summing up, using the index is cheaper if:

$$\frac{N}{64} \geq \log_{512} C + N_{>50}\left(1 + \frac{1}{1024}\right)$$

In most practical situations, the first and last terms of the right-hand side are negligible, and the question becomes whether $N_{>50}$ is less than or equal to $N/64$. To decide, we need statistical information about the credit values, usually stored in a *histogram*.

A histogram is a summary of the distribution of the values of a relation attribute, formed of a fixed number K of buckets, chosen small enough so that this summary can be stored in main memory and used by the query optimizer without incurring the cost of a disk seek. For each $1 \leq i \leq K$, bucket i contains statistical information about tuples for which the attribute value is between v_i and v_{i+1}. Thus, v_1 is the minimum attribute value and v_{n+1} the maximum value. The information stored is typically the number of distinct values in the interval $[v_i; v_i + 1)$, the number of tuples containing a value in this interval, and possibly other statistics of interest, such as the mean or median value in each bucket. A histogram can be used to estimate the number of results to a range query such as $Q_{>50}$: add up the number of tuples in buckets whose range intersects the range of the query, possibly refining the estimate for buckets that are at the boundary.

There are different ways to organize attribute values into histograms. *Equi-width* histograms partition the set of values $[v_1; v_{n+1}]$ into K intervals of the same size. This scheme is well adapted when the data distribution is close to a uniform one, but fails when the distribution is too biased. Going back to our example, assume that the minimum and maximum number of credits are respectively 0 and 1,000, but most students have credits less than 150. If we construct an equi-width histogram with 5 buckets, most of the data values are represented by the bucket $[0; 200)$, and the histogram is not very helpful to estimate $N_{>50}$. To avoid this issue, it is possible to use *equi-height* histograms, where the buckets are constructed such that the number of tuples per bucket is more or less uniform. In our example, this means that several buckets cover the interval $[0; 150]$. An estimate of $N_{>50}$ adds up the total number of tuples in all buckets whose lower bound is greater than 50, plus a fraction of the number of tuples of the bucket where 50 is contained, which yields a more precise approximation. An equi-height histogram, however, is more difficult to maintain in the presence of update operations than an equi-width histogram.

Research on histograms has aimed at proposing new ways of splitting the data values into buckets (e.g., v-optimal histograms (Ioannidis and Poosala 1995) whose frequency estimates are provably optimal for a large class of queries), at maintaining histograms when the data is updated (analogous to the problem of view maintenance), at efficiently computing histograms from the base data (mostly with the help of sampling techniques), or at building join summaries for the distribution of several attribute values, to deal with queries with multiple selection criteria. We refer to Poosala et al. (1996) for more details. DBMSs typically use both equi-width and equi-height histograms (Breitling 2005) and allow choosing between the two when tuning a database.

The Cascades Optimizer

We now explain briefly how a real query optimizer might generate a number of possible query plans, using logical optimization rules and available data access methods, evaluate their cost and decide on the plan to run. One of the main problems is to avoid a combinatorial explosion that would result in trying to apply all possible transformations. We take the example of the Cascades query optimization framework (Graefe 1995) that was intended as the basis of the query optimizer of Microsoft SQL Server. All logical optimization rules (*transformation rules*) and data access methods (*implementation rules*) are described as algebraic rewritings of a query plan. Each query plan is associated with its cost, which can be computed from the costs of the sub-plans. Cascades optimizes a query in a top-down manner using *memoization* to remember the optimization decisions for each encountered sub-query plan. When optimizing an expression, the system first considers if this expression (or one that is "similar enough") has not already been optimized, and, if so, directly uses the result. Otherwise, transformation rules and implementation rules applicable at the top-level are applied, using the guidance of the predicted cost of the resulting query plan, They also use heuristics that bias the exploration strategy, and *promises* for each rule that can be used to condition its application depending on previous and subsequent rule applications, in a goal-driven manner. Sub-expressions of the query are then optimized one by one, following the top-down process.

Data Indexing and Storage

Once a query plan has been selected by the optimizer (see Fig. 10.2), it is executed, using the indexes and data storage structures referred to in the plan (remember that the different methods of accessing the data have been considered and their cost estimated when optimizing the query). Most DBMSs index and store the data in a similar way: ordered datatypes are indexed using B-trees or B^{+-}trees (Bayer and McCreight 1972; Comer 1979), unordered datatypes with hash tables (sometimes dynamically maintained (Fagin et al. 1979; Litwin 1980)), and whole tuples are stored either in the nodes of the B-tree or B^{+-}tree index for their primary key, or sequentially sorted along their primary key, aligned with disk pages. A large body of research was dedicated to improve and build variants of these classical data structures, widely used for generic database applications. We present now research on alternative indexing and storing strategies that have been widely used for specific kinds of data: multidimensional indexes to efficiently retrieve objects based on their locations in Euclidean spaces, column stores that organize the data column-by-column instead of the traditional row-by-row storage strategy, and stream databases where data is not stored at all but queries are processed continuously as data arrive.

Multidimensional Indexes

B-trees and their variants are used for indexing linearly ordered data (integers, character strings, etc.) and efficient processing of point or range queries over the indexed attribute, i.e., selections like $\sigma_{credits=50}$ and $\sigma_{credits>50}$. Imagine now that the *Students* table contains two additional columns, *lat* and *long* that respectively contain the latitude and longitude of their home address, and that we want to retrieve the list of students who live less than 10 km away from campus. We could index these two attributes with a B-tree, but B-trees are not well adapted to this kind of query, and the best we could do would be something like computing the minimum and maximum latitudes covered by the 10 km radius, retrieving all students with latitude in this range, and then checking for each of them whether their combined latitude and longitude fall in the 10 km radius. Indexing structures have been proposed to better deal with such queries, such as quadtrees or R-trees.

Let us start with quadtrees (Finkel and Bentley,1974). Assume that latitudes and longitudes of student home addresses form a multiset of two-dimensional points. A quadtree divides a bounded region of 2D space (say, a square containing all points) into subregions in the following way. The square is divided into four squares of equal size, and each subsquare is divided again, recursively. A square is not divided further when it contains less than K points, for a fixed threshold K. This division of the 2D space naturally defines a tree of arity 4: the root of the tree is the whole region, the children of a node are the four subsquares of this node, and leaves point to the $\leq K$ points contained in the corresponding region. The construction and maintenance of such a structure is relatively easy. To answer the 10 km radius query, we retrieve, by a top-down browsing of the quadtrees all leaves that intersect the 10 km disc. Points in leaf regions entirely contained in the disc are returned immediately, while points in leaf regions that only partially intersect the disc are filtered one-by-one.

Quadtrees usually provide an efficient way of answering a geographical query, but they have one weakness: if the distribution of points is too biased (e.g., if many students live on campus accommodation, very close to each other), the tree may become quite unbalanced. Furthermore, in contrast to B-trees, the arity and depth of the tree are not optimized with respect to the number of disk pages accessed while searching the trees. R-trees (Guttman 1984) provide a solution to both of these problems. Again, the space is divided into a number of rectangular regions that are organized in a tree, but now the regions may overlap, are of arbitrary size and shape (though the region corresponding to the parent of a node is still a proper superset of the region of this node), and the tree is organized like a B-tree, with a large arity that is computed so that each tree node fits into a single disk page. Algorithms for searching and updating the R-tree are move involved than for quadtrees and directly inspired by their counterparts in B-trees. Again, answering the 10 km radius query means retrieving all leaves that intersect the 10 km disc, and subsequent filtering of the points contained in the leaves. R-trees generalize more easily to arbitrary dimensions than quadtrees, for which the arity is necessarily an exponent of the dimension.

Quadtrees and R-trees are widely used in geospatial extensions of DBMSs (Kanth et al. 2002), to index multidimensional data. They illustrate how the database community has proposed efficient data structures when new datatypes and applications of database technology appeared.

Column Stores

The main idea of column stores (Stonebraker et al. 2005) is simple: instead of storing tables row-by-row, with a whole tuple stored contiguously, they store tables columns-by-columns. Assuming again a page size of 4 kilobytes and 64 bytes for storing a tuple of the table *Students*, a traditional DBMS stores 64 tuples per page. Assuming 2 bytes for the *credits* attribute, a column store puts 2,048 values of this attribute in a single page. The interest of column stores is immediate in this numeric example: computing an aggregate of the *credits* attribute across all students, such as its average or sum, requires 32 times less disk page accesses than with a row store. Not all operations benefit of this data storage organization, however: any operation that needs access to all tuple values, such as computing a full join between two tables, or inserting individual tuples, typically requires more random seeks in a column store than in a row store. Generally speaking, applications that heavily use aggregates, statistics computation, or more generally individual attribute values rather than whole tuples, usually benefit from column stores. These applications are sometimes called *online analytical processing* (OLAP), in contrast with *online transaction processing* (OLTP) that cover more traditional database applications, such as order processing or banking. Another advantage of column stores is their ability to use more effective compression mechanisms to reduce the size of the data store, since data of a single type are stored contiguously.

Commercial (e.g., Sybase IQ, Vertica, KDB) and open-source (e.g., MonetDB) column-oriented databases coexist with traditional DBMSs. For a long time, all database-related tasks had been handled by traditional engines; the emergence of column stores might be an illustration that there is room for technologies that depend on the applications (Stonebraker 2008).

Stream Databases

Following up on the idea that classical DBMSs may not be adapted to all database management problems, we now consider the case when data is produced in such a large volume or at such a high rate that it cannot even be stored. A typical example is network data (see Chap. 7.): IP packets that go through a router of the Internet core are too numerous to be stored on disk. If one needs to query these packets (selection, aggregation, grouping), e.g., to detect potential attacks or trends in the use of the network, one needs to reverse the model: instead of evaluating various queries over a somewhat fixed collection of data, we want to evaluate a fixed set of

queries over continuously streaming data. This is the model of stream database systems, such as Gigascope (Cranor et al. 2003).

For flexibility, one would like to use a general-purpose query language like SQL to query the stream of data. Note, however, that since it is impossible to store the entirety of the stream, some queries cannot be evaluated, such as those involving arbitrary joins with past or future data. For this reason, Gigascope defines a restriction of SQL where queries need to be evaluable in a *sliding window* of fixed size. All relevant packets in this sliding window are typically stored in memory. Query optimization has very different constraints than in classical settings to avoid missing some of the packets, it is critical to reduce the amount of data kept in memory by pushing the operations with the highest selectivity as early as possible, sometimes even implementing them in the code of the network interface controller. Higher-level operations (joins, grouping, etc.) can be applied later on the buffered data. It is also possible to increase performance by partitioning the stream and have each substream handled by a different computer; this partitioning, however, must not put in two different groups packets that need to be used together to answer a given query, which implies basing the partitioning operation on the query (Johnson et al. 2008).

A number of prototypes and commercial systems for database stream management have appeared. Even more so than for column stores, their applicability is restricted to very particular scenarios: network traffic, real-time auction market analysis, etc.

Hardware and Why it Matters

We are now at the bottom of the query processing system, where the query is executed on actual hardware. Perhaps even more so than for other software, the performance of database management systems has been strongly tied to the evolution of hardware architectures. The design of cost models, index structures, storage engines in traditional DBMSs has been based on a number of assumptions on how hardware functions:

1. High cost of disk accesses, and, especially, random seeks
2. Limited amount of available main-memory
3. Mostly serial CPU instruction processing model
4. Relatively low network bandwidth

However, as elaborated in Chap. 5, hardware and network infrastructure have evolved to the point where the validity of all these assumptions can be questioned:

1. The recent advent of flash memory and solid-state disks radically change the performance of disk accesses (see below)

2. The amount of main memory available in even low-end PCs makes it possible to store database indexes and sometimes even the data itself in main memory (Garcia-Molina and Salem 1992)
3. Parallel architectures are more and more frequent, to the point where standard modern graphics processor units are able to process hundreds of parallel execution flows, which makes them suitable for some database management tasks (Govindaraju et al. 2006)
4. The network bandwidth in a local cluster can be higher than disk transfer speeds, which has the ability of making main-memory distributed DBMSs more efficient than centralized disk-based ones (Apers et al. 1992) (see the next section for a discussion of distributed databases)

These examples explain why the evolution of hardware architectures does matter for database management systems, and why research on understanding and exploiting new capabilities of hardware is an active component of database research. We illustrate with the example of solid-state drive for database storage.

SSDs vs Magnetic Hard Drives

Secondary (i.e., non main-memory) storage of data in DBMSs has mostly relied on magnetic hard disk drives. These disks are made of one or several rotating platters where information is encoded by the orientation of the magnetic field generated by localized regions, organized in concentric *cylinders* and radial *sectors*. Information is read and written with magnetic heads that hover over the platter. Reading or writing to an arbitrary region of the disk requires a *random seek*: the head of the appropriate platter needs to be positioned over the correct cylinder and then wait for the moving disk to reach the correct sector. A *sequential read* that retrieves contiguous portions of data, on the other hand, is much faster since the head can remain fixed and the data is read as the disk rotates. The order of magnitude of the seek time and read sequential data transfer rate are, for a modern disk, respectively 10 ms and 50 megabytes per second.

Since the mid-1990s, a new form of permanent data storage has appeared: *flash memory*, using *floating-gate* transistors, transistors wired on chips so as to store an amount of charge for extended periods of time. Recently, the technological advances in building flash memories have led to the commercialization of solid-state drives (SSDs for short), which are drop-in replacements for hard disk drives formed of an array of flash memory units. The absence of any mechanical parts in such drives leads to negligible seek times. Modern SSDs have read data transfer rates comparable to that of magnetic drives, whereas sequential write transfers are somewhat slower (but random writes are typically faster).

The near-absence of seek times makes SSDs particularly suitable for database applications, where it is common to read small data blocks scattered across the storage area. Recent studies (Lee and Moon 2007) show that, indeed, read performance of DBMSs can be dramatically improved by using SSDs.

However, using a traditional DBMS on a solid-state drive causes other forms of problems, related to an inconvenience of flash memories: it is impossible to update a data item in place without first erasing the corresponding block of flash memory. This means that actual writing speeds are considerably slower than expected. To avoid this issue, Lee and Moon (2007) proposes to implement updates by logging all update operations in a fragment of each memory block kept free for this purpose. Reading the current state of the data consists thus in reading the base data, and updating it in memory with the extra logged updates. When the logging area is full, the whole block is erased and rewritten. This approach, which heavily relies on the behavior of flash memory, in the same way as traditional indexing approaches heavily rely on the fact that disk seeks are costly, allows obtaining improved performance over a regular DBMS on magnetic drive, even when considering update operations.

Another important aspect of building database systems with SSD storage is to understand the precise behavior of SSDs. A number of SSDs are thus benchmarked in Bouganim et al. (2009), exhibiting some counterintuitive results. For instance, despite the lack of mechanical parts, some seek latency appears, mostly because of the overhead introduced by controlling software. Another observation is that SSDs often do not exploit the possibility of parallelizing reads and writes operations over the flash memory arrays. Some of these characteristics are likely to be transitory behavior of SSD controllers, while some others will be important in designing future database management systems.

Distributed Databases

Before concluding this section on core database research, we want to mention the important aspect of distribution in database management systems that pervades the entire query processing pipeline. We say that a database system is *distributed* when the data itself is spread over a number of computers (also called peers, or hosts) connected over a network (See Chaps. 7, 8, 9.) The role of a *distributed DBMS* is then to manage this distributed database and to "make the distribution transparent to the users" (Özsu and Valduriez 2011). There are several reasons why we might want to distribute data:

- Data may be distributed to begin with, because of organizational reasons. Think, for instance, of the human resource and sales data of a company, which might reside in different departments, possibly in different physical locations, but sometimes needs to be seen as parts of a single database, e.g., for business intelligence purposes. At the extreme, the World Wide Web may be seen as a gigantic database consisting of data distributed all over the planet, seen as a whole by applications such as search engines.
- Data may simply not fit on the disk(s) of a single computer, however large they may be. A database that records stock market transactions, or meteorological

data, for instance, may get to enormous sizes: several hundred of terabytes for the database maintained by the Max-Planck-Institute for Meteorology (WinterCorp 2005).

- Non-distributed databases provide a single-point of failure: should the computer hosting the database fail, or should the number of data access exceeds what the database management system is capable of handling access to the entire database would be lost. Conversely, if the data is distributed, the load is divided between all peers, and a failure of a single host only affects part of the data. Availability can even be guaranteed to some extent if data is *replicated* over several peers of the network.
- It is possible in some cases to distribute data to improve the efficiency of query evaluation. We have already mentioned that a distributed database with data in main memory may be more efficient than a traditional local database with data on disk. Even when data is stored on disk, distribution allows parallel processing of a query. Recall that queries of first-order SQL can be evaluated in constant time on a parallel machine, which means that first-order SQL query evaluation can be very efficiently run in a parallel manner.

How data is distributed over the network depends on the reason data is distributed. When distribution is inherent in the organization of the database, there is no choice and it is often the case that data is stored in a heterogeneous manner and needs to be *integrated*, as we explain further in this chapter. When data is distributed for size, reliability, or optimization reasons, different data distribution strategies can be selected. Most commonly, relations are either *horizontally* or *vertically fragmented* (Özsu and Valduriez 2011). In horizontal fragmentation, a table is partitioned along its tuples, and groups of tuples are stored in different peers of the network. In vertical fragmentation, the partition is made according to the attributes of tuples, and each peer stores a subset of the attributes of each tuple, as well as its primary key. This storage choice is reminiscent of the distinction between row stores and column stores, and similar tradeoffs arise. Another point of interest is the network architecture used, which can range from centralized settings where a master host, connected to a number of slave hosts, acts as an entry point to the database, to distributed tree structures or fully distributed models such as distributed hash tables over peer-to-peer networks (Abiteboul et al. 2011).

A number of traditional database problems, such as query optimization or transaction management raise radically different challenges in a distributed environment. In some cases, this has led to relaxing some of the constraints traditionally imposed by relational DBMSs. This trend, sometimes dubbed the *NoSQL* movement, has resulted into distributed data and computation systems that do not support ACID transactions and have limited expressive power, but very high efficiency on extremely large collection of data, such as the MapReduce framework (Dean and Ghemawat 2008), extensively used by companies such as Google to process petabytes of data a day.

For an in-depth discussion of distributed database systems, we refer to the textbook (Özsu and Valduriez 2011).

Research on core database technology covers a large spectrum of areas, from logics to systems and optimization issues, even up to the benchmarking of modern hardware. We now move to a different vein of research, to extend the main approaches that have made the success of relational databases (abstraction, algebraic representations, etc.) to cover other applications and functionalities.

Extending Database Functionality

We have stressed that the relational model is a natural evolution point as data management systems increase in their abstraction – hiding from the programmer or end-user details of the physical layout of data and the implementation of queries. But in some ways the relational model is low-level: the data model (at least, as visible to the data definition language) imposes quite a few restrictions, including allowing only a fixed set of simple data types as attributes of a tuple, requiring the data developer to spend time "breaking down" information into small components. Indeed, this is a basic part of the philosophy of the relational paradigm towards data design. In addition, the set of features in a relational schema – particularly with regard to integrity constraints – are very limited compared to the kinds of semantic restrictions that one may want to express about real-world data.

Much of the research in the database community has revolved around extending the mathematical foundation of database systems to be less "low-level" (in data model). Another direction has been to look at richer data definition languages, even within relational databases. A closely-connected topic is the ways of building up larger datasets from components – data integration. Some of these extensions have been pursued while trying to preserve the relational approach in its entirety. For example, in the case of query languages for XML documents, database research still takes a declarative approach, compiles into an algebra, and applies rule-based optimization. In other cases essential features of the relational paradigm are jettisoned completely. We will take a quick look at each of these general lines of research within this section.

Data Design

The relational database model is built on a very simple data structure, a table where each cell contains a simple type. Nevertheless, it was seen that one can represent the information in many applications using relations, by breaking down more complex structures into tables. But exactly how should complex data be translated into tables?

The major challenge is that there are many ways of representing the same information. In the university example, we had a *Students* table including *id*, *first*, *last*, and *credits* (the student's current credits), and an *Enrollment* table that

included *id*, *course*, and *credits* (the course credit value for this student). But one could also have one large table *StudEnroll* that had all of the previous attributes. The second possibility seems odder, but can we say that it is worse?

The subject of *data design* originated very early in database research; its goals included:

- Capturing a notion of two schemas representing the "same information"
- Formalizing the notion that one schema is better than another
- Providing algorithmic techniques for getting to a good schema

These goals could be seen from the same two-pronged perspective that defines database research as a whole. On the one hand, in attempting to define the notion of "information equivalence", database research was exploring the "theory of information" in a very grand sense. Certainly an insight into what constitutes the same information content within data would be significant even if it was not accompanied by effective methods. On the other hand, data design research had a pragmatic goal of offering advice to database designers on how to create and maintain database schemas.

It should be clear that such a project must limit its scope in some way. First of all, there are many human factors in determining what a good schema is – database research cannot say if one column name is better than another, or whether it is better to store the yearly salary or the monthly salary of employees (since clearly one can derive one from another). Thus the best we can hope for is that computer science research could identify certain designs as being inferior or superior to others: we cannot hope to identify a unique "best design". Second, such a process must take as input some information about the semantics of the database, not just that which is captured in standard table meta-data. For example, if we only know that we have a table named *StudEnroll*, including columns *id*, *first*, *last*, *course*, *creditsObtained*, and *creditsCourse*, but with nothing about its meaning, we cannot identify that there is any shortcoming. Data design thus starts with a description of the "semantics of information to be stored" in some richer data model (these may include nested tables, lists, sequences, or other higher-level structures), and then gives a method for translating to a relational database schema. *Entity-relationship* diagrams represent one such formalism for high-level data description; there is a simple algorithm for generating a relational schema from an entity-relationship diagram. More powerful modeling languages, such as UML, can also serve as a starting point.

The most well-developed theory of "better design" has looked at simpler languages for describing the semantics of information. Most of the algorithmic results work in a simple modification of relational data definition languages, in which information is described using a set of tables plus integrity constraints. These include SQL key and foreign key constraints, as well as more powerful constraints. The paradigmatic example uses *functional dependencies* as the constraint language. A functional dependency states that a subset of the columns determine other attributes of the table. In the *StudEnroll* example, we have that the value of *id* determines the values of *first* and *last*.

The standard theory contributes a notion of "better schema", formalizations of "information equivalence", and a method for going from bad schema to a better one.

The problem with the schema above can be identified formally by the presence of a functional dependency (*id* implies *last*) that does not follow from a key constraint (*id* is not a key, it is repeated in multiple rows): such a dependency implies that information in some columns in the row is redundant, and hence will be repeated many times. If physical storage reflects the repetition in the table structure, then this will clearly lead to performance issues, as well as extra infrastructure needed to maintain consistency during updates. The difficulty is summarized as: a piece of information should only be represented in one place, and to change it one should only need to modify in one place. A schema that includes functional dependencies is said to be in *Boyce–Codd Normal Form* (BCNF) if all functional dependencies follow from key dependencies.

What does it mean for two schemas to have the "same information"? One well-studied definition is that a schema B is a *lossless-join decomposition* of schema A if tables in A can be obtained by joining projections of tables in B. The schema consisting of tables *Students* and *Enrollment*, with the obvious key dependencies, is a lossless-join decomposition of the *StudEnroll* table; the lossless-join property states that *StudEnroll* can be exactly recaptured using the join $Students \bowtie Enrollment$.

Algorithms exist (Codd 1975) for automatically finding a lossless decomposition of an arbitrary schema into a BCNF schema. Normalization can be seen as a design methodology; start with an initial design – for example, one reflecting the user interfaces that end-users would like to see. Then continue to decompose until a normal form schema is obtained. The original tables can be re-captured either as ad-hoc queries, or as materialized or virtual views. If the un-normalized tables are materialized, many of the space benefits of normalization are lost. But even then the benefits for software infrastructure will remain; updates will need to be specified only on one table, and any update of redundantly-stored information will be done automatically.

From a theoretical point of view BCNF decomposition is the most basic example of *normalization theory*. Normalization has been considered for richer schema languages, including a number of other kinds of integrity constraints, such as multi-valued dependencies (Fagin 1977) and join dependencies (Fagin 1979). Stronger notions of information preservation have also been considered, such as being able to enforce all of the original integrity constraints on the decomposed schema using simple key constraints. In each case, the theory investigates whether or not equivalent schemas can be found for any schema in the data model. For example, a basic positive result in the theory is that for any schemas consisting of rich collections of integrity constraints (functional dependencies and multi-valued dependencies), one can find a lossless decomposition that requires only key constraints (Fagin 1977): the corresponding decomposition is said to be in Fourth Normal Form – a stronger normal form than BCNF. A sample negative result in the theory states that there is a schema S consisting of very simple integrity constraints, such that there is no lossless decomposition of S into schema S' in which key constraints on S' suffice to enforce all integrity constraints in the original

schema. This result motivates enforcing weaker criteria on the decomposed schema (such as Third Normal Form (Zaniolo 1982)).

Normalization theory is an extreme example of the dual role of database research. On the one hand, a basic understanding of the virtues of normalized tables is considered essential for data designers and database consultants. On the other hand it represents a broad investigation into the meaning of information.

Advanced Data Definition

From Stored Data to Virtual Data

Data definition languages represent an important component of the relational model, playing a role analogous to type systems in general-purpose programming languages. The basic DDLs describe the attributes and attributes types in a set of tables and give integrity constraints that encode restrictions on the possible instances of the tables, along with relationships that must hold between tables. One kind of "relationship" is a foreign key constraint, mentioned already. Another extreme example of a relationship between tables is when one table is completely determined by another. This is exactly the case of *view definitions*, previously discussed.

Tables with foreign key constraints between them can still be updated independently, and a collection of such tables generally have the same "status" as a representation of the real-life facts to be stored. In contrast, the use of view definitions – whether materialized or virtual – requires a distinction in the kinds of tables that a database manager knows about, into those that are "basic" and those that represent derived data. Hierarchies of derived data can then be defined with views defined over views.

A particular use of virtual views is in *data integration*. Suppose we have several different database schemas $S_1 \ldots S_n$ aiming at storing similar information. We wish to create a single unified interface to the data. Our first step is to come up with a single *global schema* S that can represent all information in any S_i. After that, we can give a logical definition of the global object in terms of the data I_i for each S_i stored on each local source. The single integrated database is a prime example of a *virtual database* – it can be given a precise definition, in terms of existing data, but it need not exist on any source.

How can we unambiguously define the integrated database? The simplest way is to create a query Q that takes instances $\vec{I} = I_1 \ldots I_n$ over $S_1 \ldots S_n$ and outputs a *global view instance* I over the global schema. The global instance I need not ever be materialized explicitly; instead the backend of the integrated interface generates queries to the appropriate S_i in response to queries over S.

The approach outlined above, often referred to as *global-as-view* (GAV), is conceptually straightforward, though performing the query-generation at runtime can be problematic. But the simplicity is misleading: when n is large a query Q

describing S in terms of the S_i may be difficult to write, and possibly impossible to evaluate. Furthermore, maintaining Q as the sources are modified is difficult, since its rewriting in terms of the source schemas may not even be human-readable.

More Complex Virtual Databases

An alternative is *implicit specification* of the global instance, as a data source I over the global schema S that satisfies various constraints with respect to the local sources $\vec{I}$. The most common approach to doing this, known as *local-as-view* (LAV) (Lenzerini 2002) describes I by giving constraints of the form $I_i \subseteq Q_i(I)$.

Given instances $I_i{:}i \leq n$ for the input schema, constraints of this form do not determine a unique database I, but rather a collection $Sol_V(\vec{I})$ of instances satisfying the constraints.

Although we cannot talk about "the integrated view", we can still make sense of querying an integrated view: the result of a query Q on the view is taken to mean the intersection of all $Q(D)$ for D in the collection $Sol_V(\vec{I})$. This set of results, often called the *certain answers of* Q, is equivalently seen as the set of facts of the form $t \in Q(I)$ that are logical consequences of the input data $\vec{I}$ and the view definitions relating $\vec{I}$ to an arbitrary solution I.

As an example, consider a data integration system that defines a virtual relation *Enrollment* with attributes *id* and *course*. One local source may have a stored relation *StudentIds* which has a single attribute *id*, while another might have a relation *Courses* with attribute *course*. The global view is related to the local sources by the mappings: $StudentIds = \pi_{id}(Enrollment)$ and $Courses = \pi_{course}(Enrollment)$. This defines the collection of enrollment tables that project onto the sources in the expected ways.

The advantage of the LAV approach in specification is fairly evident: specifications can now be much smaller, since they relate only two instances at a time. There is an enormous gain in modularity, since when a new source is added one must only write a new set of constraints involving only I and that source, and changes to the schema of a source I_i require only modifications to constraints involving I and I_i.

The disadvantage is also obvious: since the collection of instances satisfying the constraints is generally infinite, it is not clear how to calculate the tuples that lie in $\bigcap_{D \in Sol_V(\vec{I})} Q(D)$ at all, much less how to calculate them efficiently. A fundamental result is that for LAV views the certain answers for positive queries (an extension of the conjunctive queries) can be calculated efficiently in the size of the data (Levy et al. 1996). In fact, one can create a single view instance I that is "universal", in the sense that performing a conjunctive query on I gives the certain answers of Q with respect to the source instances $\vec{I}$ and the view definitions. One forms the universal instance I by simpling throwing in "dummy witnesses" that are implied by the view definitions.

For example, the view definition may state that for the integrated view I with attributes a, b, c, we have the requirement that $I_i \subseteq \pi_{a,b}(I)$ where I_i is a given source. Then for any tuple (a_0, b_0) in I_i, a solution I must have some value (a_0, b_0, c) in it. The universal solution is formed by choosing a distinct c for each such (a_0, b_0).

The ability to form universal instances gives an algorithm for determining the certain answers that is polynomial in the size of the data sources $\vec{I}$. This shows that implicit ways of defining virtual databases can still be efficiently implemented.

Integration vs. Extraction in Commercial Systems

The notion of data integration above is to create a virtual interface to data in different formats that may be accessed by external sources as if it were a centralized database. An alternative is to extract the data from the diverse sources and materialize the data.

One form of integration system, federated databases, has a fair amount of commercial support. For example, there is support in IBM DB2 and Microsoft SQLServer for creating views that refer to external databases. Federated database managers are not limited to relational databases (or to views defined via queries) but can encapsulate access to pre-relational or proprietary data, via hand-coded stored procedures.

Extraction-based approaches, in which explicit or implicit derived databases are materialized, are also supported by many commercial systems. IBM's DataStage product includes support for extraction based on implicit view definitions; most of the commercial usage of such systems, however, is done via manual coding of transformations.

More General Implicitly Specified Databases

There are many other formalisms that have been devised for defining virtual databases. Many extend the general contours of the LAV approach: the virtual database I is defined by a set of constraints that hold between it and stored data instances I_i. *Source-to-target dependencies* are one example of such constraints (Fagin et al. 2005). Care must be taken in both the constraints and the queries one is allowed to pose against the virtual database. Calculating the certain answers requires solving a satisfiability problem, and satisfiability is known to be undecidable for many query languages, including the relational calculus.

Another example of an implicit database formalism is that of deductive databases; in this case, a virtual database is defined by giving facts that it contains as well as axioms on the database itself, rather than on its relationship to other databases. Answering a query against the virtual database is again defined in terms of deduction: a fact is in the query result if the axioms and facts of the virtual database imply it. These axioms must be of restricted form in order for deduction to

be decidable. For example, Horn clauses, of the form $\forall \vec{x}.R(\vec{x}) \rightarrow S(\vec{x})$, are one popular formalism for deductive databases.

Ontologies and Databases

Our last example of an implicitly specified database comes from *ontologies*. An ontology consists of a collection of facts coupled with axioms written in a restricted fragment of logic. The collection of facts is not taken to include all true statements about the real-world relation, but only a subset. The axioms are interpreted to hold not over the database of facts, but over the entire universe.

In the university example, we may have facts listing certain entities as being math professors, certain entities as being students, and some facts about which students are advised by which professors. For example, we may have a fact that student Bob Jones is advised by Rob Smith: *Advises*(Bob.Jones, Rob.Smith). We have an additional axiom stating that every math student is advised by a math professor. In the notation of description logics, this would be written: $\mathit{MathStudent} \subseteq \exists \mathit{AdvisesMathProf}$. This axiom is not treated as an integrity constraint on the set of facts: if it had been, then it would fail if Rob.Smith is not listed as a physics professor. Instead it is used to derive new facts. A query asking for all mathematics professors will then return Rob.Smith, since the axiom coupled with the fact base implies that Smith is a math professor.

As in the case of LAV integration, an ontology gives an incomplete description of a collection of data. Answering a query against an ontology is defined again in terms of certain answers: for a query Q, we return all the tuples t such that the database of facts and axioms derives that t is an answer to Q.

The exact formalism used for the axioms is restricted so that the derivation of facts can be effectively decided. A standard has emerged over a set of ontology languages, based on description logics (Baader et al. 2003) – a limited logical language in which all input relations must have at most two attributes. The queries must also be restricted, usually to be positive SQL queries without aggregation. Even with these restrictions, the complexity of the decision procedures is high: even for the simplest language, consistency of a fact is PSpace-hard (Schmidt-Schaubß and Smolka 1991) in the ontology. Nevertheless, the complexity for a fixed set of axioms, varying only the set of facts, is often manageable. Indeed, for many ontology languages, the certain answers can be determined using database methods: for any fixed query Q and ontology O based on axioms within the family, we can generate a first-order SQL query Q' that returns the derivable answers when evaluated on the facts. This is true of the commonly used DL-lite family (Calvanese et al. 2007), and also of more recent extensions (Calì et al. 2010) that subsume ontology languages and LAV-like data-exchange formalisms.

Ontology languages have been standardized by the World Wide Web consortium. The resulting family of languages, OWL (Horrocks et al. 2003) have a number of prototype implementations, in addition to limited commercial support. The approach via rewriting to a database language has been implemented in several research

systems, particular QuoOnto (Acciarri et al. 2005). The main issue in these approaches is that ontologies may have thousands of axioms, and even an efficient translation may yield a query of size larger than current database managers can handle.

New Models: Complex Objects

Programming languages have available a rich collection of data types – they can form lists, associative arrays, vectors, and object classes that contain fields that may themselves be complex structures.The type system of relational databases in comparison is quite impoverished. Relational DBMSs manipulate "tables" or "relations" – from the theoretical point of view, a relation is a set of tuples, with each tuple being a function taking a column within a predefined set of column names to a datatype that is associated with the column. When we compare this to arbitrary programming language datatypes, we can see several dimensions in which they are limited: Relational tables are *homogeneous* – the data type within a table cannot vary row-by-row. They are also *flat*: although the columns of a relational database can have arbitrary built-in scalar datatypes, they cannot have any internal structure – or at least, no internal structure that can be referred to in the query language. Finally, they have no order and no duplication.

Of course, some of the use of complex data structures in programming languages is related to their more general mission – arrays and lists play a role in many fundamental algorithms, which are not intended to be implemented within a database manager. Still, much application data does have a rich internal structure, and relational databases often force users to use a structure that does not reflect their natural level of abstraction. This "impedance mismatch" has caused considerable concern in the database community, particularly since the move to relational database managers was prompted by a desire for greater abstraction.

We list these as limitations of the relational model, as espoused in papers and textbooks. Commercial database systems have worked from the beginning in a model that does not abide by these limits. They allow some limited heterogeneity by allowing certain cells of a row to be optional. The SQL query language supports this via primitives that can test for the presence or absence of a value for a given column. Although they generally require stored tables to be duplicate-free, they allow query results to contain duplicates, and SQL allows a query both to filter based on the position of a row in a result and to specify the ordering of the results. Nesting is supported as part of aggregate functions.

Still, the SQL extensions to support these are ad-hoc; the operators that support them cannot be freely composed, and some of them are available only at top-level. Researchers have tried to fill the gap between the theoretical model of pure tables and the SQL data model in practice, by developing a formal model that incorporates richer data types. The general goal is to follow the paradigm of the relational model: define a query language that (a) corresponds to a "natural" logic; (b) defines only

polynomial-time queries; (c) can be translated into an algebra – defined loosely as a variable-free formalism.

Nested Structures

In the relational model, the basic object is a set of tuples. In the *complex-value data model* we can iterate tuple formation and relation formation, generalizing schemas to types that can combine aspects of both relations and tuples. Data types are build up from a given set of scalar types via tuple formation and table formation:

if $t_1 \ldots t_n$ are types, and $a_1 \ldots a_n$ are names, then there is a new type $\{ a_1{:}t_1 \ldots a_n{:}t_n \}$ whose instances are tuples, where a tuple consists of functions taking each a_i to an element of t_i;
if t is a type, then there is a new type $Set(t)$ whose elements are finite sets of objects of type t.

A database schema will then consist of a collection of objects of distinct types. Normal tables can be thought of as very special cases, of the type $Set(\{ a_1{:}t_1 \ldots a_n{:}t_n \})$, where t_i are scalar types.

Some special cases of the data model restrict the ways in which tuple formation and table formation can alternate – when these type-formers are required to alternate strictly (thus disallowing, e.g., a tuple whose attributes are tuples), the objects in this model are referred to as *nested relations*. At the query language level, the most well-studied proposal is *nested relational algebra* (NRA), defined initially for the nested relation model. NRA contains new operators for both navigating a complex-valued structure and building new structures. It includes the identity mapping on schemas as a basic query, and also the relational algebra operators product, renaming, and projection, extended to the nested case in the obvious way: for example, $\pi_{a_1 \ldots a_m}$ is a query on objects of type $\{ a_1{:}t_1 \ldots a_n{:}t_n \}$ returning objects of type $\{ a_1{:}t_1 \ldots a_m{:}t_m \}$. Selection can be extended to nested relations by allowing selection conditions to include not only equalities of two top-level attributes but also identifications of scalar attributes nested within them.

The main new language feature is for nesting and un-nesting, The nesting operator is closely related to the `GROUP BY` construct of SQL. It is parameterized by a datatype t of the form $Set(\{ z_1{:}t_1 \ldots z_n{:}t_n \})$ where $z_1 \ldots z_n$ are attributes, and a subset of the attributes $z_1 \ldots z_k$. Given a set of tuples, it returns a set of nested tuples, where there is one nested tuple for every set of tuples that share the same values for $z_1 \ldots z_k$, with each nested tuple having attributes containing the attributes $z_1 \ldots z_k$ with their common values along with a new attribute containing the set of values for $z_{k+1} \ldots z_n$ obtained for this group. Un-nesting acts as an inverse of the above operation, taking a set valued attribute and pairing it with all distinct values of the projection of an additional collection of attributes.

A related language with the same expressiveness is based not on nesting and un-nesting but on adding a mapping operation, which "applies a query pointwise". One formalization of this is using an operator that takes as input a query $Q(x)$ taking

objects of type t to objects of type t', along with a query Q' creating an object of type $Set(t)$; the result is a "set comprehension" $\{Q(x)|x \in Q'\}$ that applies Q to all elements of Q', producing a set of t' objects. A variant of this combines application with "flattening": the query Q being applied in this case produces $Set(t')$ objects, and the operator applies it to each element produced by Q', unioning the results to form an object of type $Set(t')$. Such a family of languages was defined in this way in Tannen et al. (1992), parameterized by a signature of basic operations, under the name *monad algebra*. In addition to the "flat mapping" operator, monad algebra has the basic operations of the λ-calculus (see Chap. 2, Appendix 2) along with operations for pairing and projection, union, and singleton-formation. Tannen et al. (1992) shows that when either a difference or an equality operator is taken as a primitive, the resulting language captures NRA.

How do the languages above fare in meeting the desiderata of query languages? There is strong evidence that the language is not "too expressive": queries are polynomial-time, and the Boolean queries expressed by the language are exactly those expressed in relational algebra – this is the *conservativity theorem* (Paredaens and Van Gucht 1992). An extension (Wong 1996) shows that it is never necessary to build up sets whose nesting depth is bigger than the combined depth of the input and output.

Is the language expressive enough? Evidence in the affirmative is that there are a number of languages that have equivalent expressiveness to it. In addition, if we look at natural representations of nested relations as flat relations, we find that not every relational query on representations is expressible in NRA.

Adding Support for Duplication

What about adding support for tables with duplicate rows, or sequences? In the absence of nesting, it is easy to simply re-interpret the relational algebra on bags; for example, the difference operator can be interpreted to subtract multiplicities of occurrences of tuples. It is more challenging to arrive at a query language handling nested bags, as is needed to model aggregate operators in SQL. The approach of the monad algebra can be easily modified to deal with a data model where the set type-former is replaced by a multiset or list type-former. The pointwise application operator is adapted to bags or list in the obvious way – a query is applied pointwise, preserving multiplicities in the bag case. For bags an extension of this type was formalized by Libkin and Wong (1997) and given the name Bag Query Language (BQL). Related languages are given in Grumbach and Milo (1993). BQL certainly satisfies some of the desiderata of the relational paradigm: queries are given algebraically (basically, by definition), and queries are in polynomial-time, in fact in LOGSpace (Grumbach et al. 1996)

What is the relationship of the BQL language to logic? One major difficulty encountered is in selection. In analogy with selection in relational algebra, we would like to be able to check if two bag- or list-valued attributes are the same. Set equality can be expressed as bi-containment, which requires checking

membership of every element in one set in the other. For any fixed nesting, this check runs bottoms up with a check of membership on values, and hence can be expressed using a fixed alternation of the quantifiers $\exists$ and $\forall$ in first-order logic. In the case of bags, equality requires that the multiplicities of each element are the same, which requires some form of counting. In fact, in any of the basic bag query language, one can express basic cardinality constraints on flat structures – e.g., that the in-degree of a binary relation is the same as its out-degree – by using bag creation followed by a bag equality test.

So counting is somehow inherent to a bag query language. The expressiveness of BQL over flat bags can be characterized using an extension of relational algebra with arithmetic (Libkin and Wong 1993). Although this is arguably not a standard-enough logical language to say that BQL is "canonical", the characterization is useful for showing that certain queries cannot be expressed in BQL.

More Powerful Languages

The complex-value models above are analogs of the relational algebra and calculus, which capture restricted classes of the polynomial-time queries. But what are the analogs of relational languages with recursion? For example, in the relational setting, the language of *least fixed-point logic* captures exactly the polynomial-time queries, assuming the existence of an order. Abiteboul and Beeri (1995) defines an extension of the nested relational algebra with a powerset operator, and shows that it is sufficient to define recursive operators. Gyssens and Gucht (1992) and Suciu (1997) define more limited recursive extensions of nested relational algebra; Suciu shows that they have a conservativity property with respect to fixpoint logic over relations, while Gyssens et al. (2001) shows that the two languages have equivalent expressiveness.

We motivated the complex-value model via the impedance mismatch problem between databases and programming languages. But nested structures are only one feature of modern programming language type systems. One additional feature is that of pointer or reference types. Relational database attributes can model pointers, but not the ability to create new objects (or new references) dynamically in fixpoints. Abiteboul and Kanellakis (1998) takes the natural step of considering nested languages with both recursion and object creation. The resulting language is shown to be able to define arbitrarily complex database transformations. Seen from the limitative philosophy of the relational paradigm, this shows that recursion and pointer creation is simply too powerful a combination.

Data Design for Nested Structures

DBR has also been concerned with extending relational *data design* to complex objects. At the level of theoretical analysis, there has been considerable work on defining dependencies on nested structures and studying their interaction

(Hartmann et al. 2006). Normal forms have been defined, which represent schemas for nested objects with "less redundancy" (Ozsoyoglu and Yuan 1987; Mok et al. 1996; Tari et al. 1997; for a comparison, see Mok 2002).

Industrial Support and DBR for Complex Objects

In the 1990s, support for richer programming language types was seen as the natural evolution point for commercial database systems. It seemed clear that both the relational model and relational DBMSs would be supplanted by object-oriented counterparts. Vendors made two practical cases for objected-oriented systems. The first emphasized the software productivity increase obtained by diminishing the "impedance mismatch" between program objects and storage. The second was based on performance: by retrieving data object-at-a-time, rather than relation at a time, an object DBMS could improve performance in a way similar to the benefits of set-at-a-time over tuple-at-a-time.

But while object-oriented features are found in most database managers today, there is no convergence on an object-oriented paradigm for data management that replaces relational databases. DBMS software vendors have taken several distinct approaches to merging objects and databases.

- Some products add persistent storage to an object-oriented language, emphasizing integration with the type system of the language rather than faithfulness to the features of data management software. VOSS, for example, adds persistence and transaction support to the object-oriented language SmallTalk.
- Other products build a standalone object database management solution with several programming language interfaces, supporting both navigational access (following pointers) as well as an object query language. The language OQL (Cattell 1997) was proposed as a standard object query language for object databases (ODBMSs), corresponding to an extension of nested relational algebra with powerset of (Abiteboul and Beeri 1995) and introduced with the O_2 ODBMS (Deux 1990). The ODBMS Versant supports a variant of OQL, as well as a proprietary query language VQL.
- *Object-relational* database systems (ORDBMSs) build standalone database products on a more evolutionary approach, adding on object features to relational systems. A key for ORDBMS products is extensibility: they give the developer the ability to define new types and functions; some of them (notably PostgreSQL) give ways of extending the optimizer. The SQL-99 standard incorporates many features of object-relational systems.
- Object-relational mapping tools, such as Hibernate, provide support for storing programming language objects in relational tables. These tools work on top of a third-party relational database, providing languages for programmers to define mappings between objects and relations, and runtime APIs that implement

transformations on objects by translating them to calls to the DBMS. Some of these tools support their own object query languages.

Object-relational mapping tools are in widespread use, particularly for Web development; however, they do not replace relational DBMSs, but only supplement them. ORDBMs have had broader commercial success than ODBMSs: Postgres and its successor PostgresSQL are heavily-used open-source database products that embody many ORDBMs features. All major commercial vendors claim some support for ORDBMS features, but with wide variations: none of them support the full SQL-99 standard. Since the least common denominator capability over all of these systems consists of relational database management, it is difficult to say that the era of object systems has come.

The connection between the languages proposed by ORDBMS products and those proposed in DBR is radically weaker than for relational systems. The ORDBMS standard query language SQL-99 has little resemblance to the algebras proposed for complex objects. When we turn to object-oriented database design, we find that the gap between DBR and practice is even wider: not only are the normal forms for object-oriented databases not applied in designing ODBMS and ORDBMS schemas, the normal forms are barely known outside of the research community.

XML and Tree-Structured Data

XML data management arose as an application several years after query languages for complex values and objects appeared. In querying XML documents, one notices several features that were related to complex-value models. And it is in XML that the goal of extending the relational paradigm *en masse* to a richer set of datatypes has had the most commercial and theoretical success.

An XML document has many of the properties of a list-oriented model, an ordered variant of the bag models discussed earlier. In fact, documents could be coded as nested lists. Ignoring attributes for a moment, a document consisting of a root node with tag A and children $C_1 \ldots C_n$ could be represented as a nested list whose first element is a singleton set representing A (e.g., using a one-attribute schema) and whose next n elements are representations of each C_i.

The close connection between documents and complex values gives a motivation for a query language built on an extension of the application operator (the "functional approach") mentioned above for complex objects. Consider queries Q that take as input a variable binding – a mapping of free variables of a query to a node in a document – and which output a *nodeset* sequence of nodes in some document obtained by enlarging the input document. Given Q and nodeset O, we iterate Q on O by applying it to all members of O in sequence, concatenating the resulting nodelists to get a new nodelist. This operator is the basis for the main

operator of the XML query language XQuery. A *FLWR expression* in the XML language XQuery iterates a query Q over a list created by another expression E. For example, the XQuery query:

For \$x *In* \$*root*/*descendant*::*Prof*
Return {*Prof* / *@lastname*}

returns the lastname attribute of every professor element in a document.

A major distinction between XML and list- and bag-oriented data models is that the latter have schemas that fix the nesting-depth of the data. Since queries in these former models have inputs that are typed to satisfy a given schema, this implies that any given query must only deal with structures of some fixed nesting depth. In contrast, an XML query should be able to deal with documents of arbitrary depth. The navigation or selection component of a query language must therefore have some mechanism for searching arbitrarily far down within a document. XML query languages borrow a mechanism from modal logic, the use of *path expressions* for navigation. A path expression consists of a command to move in a certain direction within a structure. In the case of XML documents, these directions are referred to as *axes*, and they are given relative to a node in a document. The *descendant* axis, for example, refers to navigation to any descendant of a node, while the *following-sibling* axis refers to navigation to a right-sibling. Path expressions are built by composing steps, which consist of axis plus tag-filters; an expression \$x/*descendant*::A selects all descendants of the node (or nodes) associates with variable \$x that are labeled with A.

The use of path expressions to navigate XML documents was pioneered by James Clark, who developed the language XPath, later standardized by the World Wide Web consortium (W3C 1999). XPath was not intended as a query language, but as a sublanguage for selecting nodes within an XML document; it was used within a variety of other XML languages, such as the transformation language XSLT. The original language was variable-free, consisting of compositions of navigation steps and filters, where filters could be built up from existence tests using built-in functions and operators. For example, *descendant*::*Prof*[not (*child*::*Advisee*)] is an XPath query that selects all professor element nodes that are descendants of a given node, where the node must not have any advisee-elements as children. Later versions of the language added variable bindings (W3C 2007a).

We have already mentioned the query language XQuery. XQuery was developed for "database-style" transformations on XML documents – data-intensive transformations that do not require a recursion procedure. XQuery combines the node selection facilities of XPath with the FLWR iteration facility – roughly speaking, modal logic combined with function application. Consider a document which contains professors (*Prof* elements) and their advisees (*Advisee* children of *Prof* elements), and which also redundantly contains students (*Student* elements).

The following XQuery query returns a list of professors who do not advise anyone, listing as children all the students without an advisor:

```
for $x in $root/descendant::Prof[not(child::Advisee)]
    return
    <Prof>{Prof/@lastname }</Prof>
    <CouldAdvise>
    for $y in $root/descendant::Student[not(child::Advisor)]
    return $y
    </CouldAdvise>
```

The outer *for* loop iterates the variable $x over the path expression returning the collection of *Prof* nodes without advisees; the inner *for* loop iterates variable $y over another path expression returning potential student advisees. The query also makes use of element constructors (such as <Prof>. . .</Prof>) that generate new nodes – these play a role roughly analogous to nesting in complex-valued models.

What can we say about XQuery in regard to the desired characteristics of query languages from the relational paradigm? The core of XQuery defines only queries with polynomial-time data complexity (Koch 2006). The variable-free language XPath itself satisfies even better bounds, having complexity polynomial in both the query and the document (Gottlob et al. 2002a); indeed, for a large fragment the combined complexity of evaluation is linear (Gottlob and Koch 2004). In terms of limitation on its expressiveness and relationship to logic, a conservativity theorem similar to that of bag query languages holds: the Boolean queries that are expressible in the XQuery core are exactly those expressible in first-order logic with an additional counting quantifier (Benedikt and Koch 2009). Furthermore, the fragment of XQuery Boolean queries that corresponds to first-order logic has been identified ("atomic XQuery" of (Benedikt and Koch 2009)).

At the level of industrial acceptance, XQuery has been quite successful, albeit not yet at the level of SQL. While languages for nested relations have been mainly confined to academia, XQuery has been standardized by the World Wide Web Consortium (W3C 2007b) as a Web standard. It is is supported both in commercial data management systems and by XML document storage products.

Conclusions

The history of database research is strongly tied with that of database management systems: in particular, Codd's work on the relational algebra has been hugely influential in the development of systems. The history of database management systems, in turn, is filled with success stories. Oracle Corporation, historically and primarily concerned with database systems, is a company with 100,000 employees, and one of the 50 largest market capitalizations (Financial Times 2010). Most Web sites critically rely on database software for managing their content, user data, and

transaction information, using either major commercial systems such as Oracle, DB2, or SQLServer, or open-source software such as PostgreSQL and the comparatively lightweight MySQL (now owned by Oracle), very popular for simple Web sites and which has captured one third of the market since its initial release in 1995 (Creative System Design 2010).

Through advances in query optimization, indexing structures, but obviously also in hardware, the performance of database management systems has greatly improved over the years: the number of transactions per second in an update-oriented benchmark has thus grown thousandfold over the period 1985–2005 (Gray 2005). On the query side, results of the TPC-H benchmark show that in the past 8 years, the query throughput has been multiplied by 50. It is obviously difficult to determine the parts of performance increase coming from software and hardware advances, but an indication that algorithmic and data structure improvements have had important roles is that the observed price/performance reduction in DBMSs beats Moore's law (Gray 2005).

We have insisted throughout this chapter on two facets of database research: on one hand, models, algorithms, and data structures for the efficiency and effectiveness of existing database systems, and on the other hand, broadly-scoped, often theoretical, sometimes even highly-speculative, research about how data can be modeled, queried, and, more generally, managed. In some cases these lines of work show no evidence of convergence. We have seen earlier in this chapter the example of static-analysis–based query optimization. Decades of work in this area have yielded sophisticated characterizations of query hardness and algorithms for query decomposition and minimization, but they have not been applied in practice. Another example is in spatial databases – although database research in general has had impact in this area, many of the more heavily-researched theories for data models and query algebras (Paredaens et al. 1994) have not influenced practitioners.

The timeframe for acceptance of relational database systems was several years, but perhaps we should be willing to wait many decades to judge subsequent work. The past has shown numerous cases of database research that was considered disconnected from applications for long periods, which has ended up being of use in systems. One of the leading figures in database theory, Moshe Vardi, mentions the example of integrity constraints (Winslett 2006):

> The work on integrity constraints in the late 1970s and early 1980s also received scathing criticism as not being at all relevant to the practice of database systems, only to reemerge later as being of central importance. When you do an exciting piece of research, it is very hard to know whether it will be relevant to the field in the long term. This is true both for theory and for experimental work. The vast majority of theory research results will likely be forgotten, as will be the vast majority of experimental work.

There are certainly missed opportunities in both directions: theoretical results not applied in systems, and practical issues not theoretically modeled and analyzed as they should be.

But of course, DBR has to be measured also by the influence on other areas of computer science. Work on indexing has strong connections with research on data

structures and algorithms (compare, for instance, binary search trees and B^{+-}trees). Since the work of Codd, database theory has focused on the complexity of evaluating logics and the expressiveness of logical formalisms: results in this area interact strongly with research in finite model theory, descriptive complexity, and computational complexity (Chap. 15.) Work on the foundations of XML has led to a better understanding of tree automata and tree transducers, a basic subject of study in formal languages. The capacity of DBMSs in handling large quantities of data encouraged the development of techniques to extract patterns and discover knowledge from large databases, a field closely related to DBR known as *data mining*. There are many other examples: The applications of database technology to the management of Web data led to research at the border of databases, information retrieval, and machine learning; database design has links with software engineering, while distributed databases raise numerous questions within networking research.

As for further reading, we have already pointed to a number of references on specific research topics. We now refer to more general works that can be of use to pursue the study of database systems and database research in more depth.

A large number of textbooks cover the design of relational database management systems and review core database research. We mention (Ullman 1988, 1989; Ramakrishnan and Gehrke 2002; Garcia-Molina et al. 2008; Silberschatz et al. 2010), but there are many other excellent examples.

With the notable exception of Ullman (1989), these textbooks deal with database systems rather than with database theory. The main reference in database theory is Abiteboul et al. (1995), which covers foundational aspects of database query languages and data models. An earlier survey, giving a snapshot of theoretically-oriented research at the time, is Kanellakis (1990). Database theory is strongly tied to finite-model theory (Libkin 2004), and the connection between the two is highlighted in Vianu (1996).

Acknowledgments We would like to thank Serge Abiteboul, Georg Gottlob, Evgeny Kharlamov, Yann Ollivier, as well as the editor of this book, Edward K. Blum, for valuable feedback about the content of this chapter.

References

Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex values. *VLDB J.*, 4:727–794, 1995.

Serge Abiteboul and Paris C. Kanellakis. Object identity as a query language primitive. *J. ACM*, 45:798–842, 1998.

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

Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web Data Management*. Cambridge University Press, 2012.

Andrea Acciarri, Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Mattia Palmieri, and Riccardo Rosati. QuOnto: Querying ontologies. In *Proc. AAAI*, 2005.

Peter M. G. Apers, Carel A. van den Berg, Jan Flokstra, Paul W. P. J. Grefen, Martin L. Kersten, and Annita N. Wilschut. PRISMA/DB: A parallel main memory relational DBMS. *IEEE Trans. Knowl. Data Eng.*, 4(6):541–554, 1992.

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003.

Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

Catriel Beeri, Ronald Fagin, David Maier, Alberto Mendelzon, Jeffrey Ullman, and Mihalis Yannakakis. Properties of acyclic database schemes. In *Proc. STOC*, 1981.

Michael Benedikt and Christoph Koch. From XQuery to relational logics. *ACM Trans. Database Syst.*, 34(4):1–48, 2009.

Luc Bouganim, Björn Þór Jónsson, and Philippe Bonnet. uFLIP: Understanding Flash IO patterns. In *Proc. CIDR*, 2009.

Wolfgang Breitling. Histograms – myths and facts. In *Proc. Hotsos Symposium on Oracle System Performance*, 2005.

Andrea Calì, Georg Gottlob, and Andreas Pieris. Advanced processing for ontological queries. *PVLDB*, 3(1):554–565, 2010.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The L-Lite family. J. Autom. Reasoning, 39(3):385–429, 2007.

R. G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In *Proc. SIGFIDET/SIGMOD Workshop*, volume 1, 1974.

Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of System R. *Commun. ACM*, 24 (10):632–646,1981.

Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. STOC*, 1977.

Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proc. PODS*, 1998.

Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theor. Comput. Sci.*, 239(2):211–229, 2000.

David L. Childs. Description of a set-theoretic data structure. In *Proc. AFIPS*, 1968.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6): 377–387, 1970.

E. F. Codd. Recent investigations in relational data base systems. In *Proc. ACM Pacific*, 1975

Douglas Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. SIGMOD*, 2003.

Creative System Design. Databases. http://online.creativesystemdesigns.com/projects/databases.asp, 2010.

Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proc. VLDB*, 1987.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

O. Deux. The story of O2. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):91–108, 1990.

Robert A. Di Paola. The recursive unsolvability of the decision problem for the class of definite formulas. *J. ACM*, 16(2), 1969.

Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. Database Syst.*, 2(3):262–278, 1977.

Ronald Fagin. Normal forms and relational database operators. In *Proc. SIGMOD*, 1979.

Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing – a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

Ronald Fagin, Phokion G. Kolaitis, Renée Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

Financial Times. The world's largest companies. Technical Report ft500, Financial Times, 2010.

Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, 1992.

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, second edition, 2008.

Georg Gottlob and Christoph Koch. Monadic Datalog and the Expressive Power of Web Information Extraction Languages. *J. ACM*, 51(1):74–113, 2004.

Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. VLDB*, 2002a.

Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002b.

Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. SIGMOD*, 2006.

Goetz Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3): 19–29, 1995.

Jim Gray. A "Measure of transaction processing" 20 years later. *IEEE Data Eng. Bull.*, 28(2): 3–4, 2005.

Stéphane Grumbach and Tova Milo. Towards tractable algebras for bags. In *Proc. PODS*, 1993.

Stéphane Grumbach, Leonid Libkin, Tova Milo, and Limsoon Wong. Query languages for bags: expressive power and complexity. *SIGACT News*, 27(2):30–44, 1996.

Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD*, 1993.

Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD*, 1984.

Marc Gyssens and Dirk Van Gucht. The powerset algebra as a natural tool to handle nested database relations. *J. Comput. Syst. Sci.*, 45(1):76–103, 1992.

Marc Gyssens, Dan Suciu, and Dirk Van Gucht. Equivalence and normal forms for the restricted and bounded fixpoint in the nested algebra. *Information and Computation*, 164 (1):85–117, 2001.

Thomas Haigh. "A veritable bucket of facts": Origins of the data base management system. *SIGMOD Record*, 35(2):33–49, 2006.

Sven Hartmann, Sebastian Link, and Klaus-Dieter Schewe. Functional and multivalued dependencies in nested databases generated by record and list constructor. *Ann. Math. Artif. Intell.*, 46(1–2):114–164, 2006.

Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: the making of a Web ontology language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7–26, 2003.

Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Proc. SIGMOD*, 1995.

ISO. *ISO 9075:1987: SQL*. International Standards Organization, 1987.

ISO. *ISO/IEC 9075–4:1996: SQL. Part 4: Persistent Stored Modules (SQL/PSM)*. International Standards Organization, 1996.

ISO. *ISO 9075:1999: SQL*. International Standards Organization, 1999.

Theodore Johnson, S. Muthu Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. Query-aware partitioning for monitoring massive network data streams. In *Proc. SIGMOD*, 2008.

Paris C. Kanellakis. Elements of relational database theory. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 1073–1156. Elsevier and MIT Press, 1990.

Kothuri Venkata Ravi Kanth, Siva Ravada, and Daniel Abugov. Quadtree and R-tree indexes in Oracle Spatial: a comparison using GIS data. In *Proc. SIGMOD*, 2002.

Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proc. PODS*, 1985.

Won Kim. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982.

C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. Database Syst.*, 31(4):1215–1256, 2006.

Isaac Kunen and Dan Suciu. A scalable algorithm for query minimization. Technical Report 02-11-04, University of Washington, 2002.

Neal Leavitt. Whatever happened to object-oriented databases? *Computer*, 33(8):16–19, 2000.

Sang-Won Lee and Bongki Moon. Design of flash-based DBMS: an in-page logging approach. In *Proc. SIGMOD*, 2007.

Maurizio Lenzerini. Data integration: a theoretical perspective. In *Proc. PODS*, 2002.

Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, 1996.

Leonid Libkin. Expressive power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, 2003.

Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.

Leonid Libkin and Limsoon Wong. Some properties of query languages for bags. In *Proc. DBPL*, 1993.

Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *J. Comput. Syst. Sci.*, 55(2):241–272, 1997.

Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. VLDB*, 1980.

Diana Lorentz. *Oracle Database SQL Language Reference*. Oracle, 2010.

William C. McGee. The information management system (IMS) program product. *IEEE Annals of the History of Computing*, 31:66–75, 2009.

W. Y. Mok. A comparative study of various nested normal forms. *IEEE Trans. on Knowl. and Data Eng.*, 14(2):369–385, 2002.

Wai Yin Mok, Yiu-Kai Ng, and David W. Embley. A normal form for precisely characterizing redundancy in nested relations. *ACM Trans. Database Syst.*, 21(1):77–106, 1996.

T. William Olle. *The Codasyl Approach to Data Base Management*. John Wiley & Sons, Inc., 1978.

Z. Meral Ozsoyoglu and Li-Yan Yuan. A new normal form for nested relations. *ACM Trans. Database Syst.*, 12(1):111–136, 1987.

M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, third edition, 2011.

Jan Paredaens and Dick Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1):65–93, 1992.

Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. Towards a theory of spatial database queries. In *Proc. PODS*, 1994.

R. L. Patrick. IMS@Conception. *IEEE Annals of the History of Computing*, 31(4):62–65, 2009.

Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. SIGMOD*, 1996.

J. A. Postley. Mark IV: evolution of the software product, a memoir. *IEEE Annals of the History of Computing*, 20(1):43–50, 1998.

Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. WCB/McGraw-Hill, third edition, 2002.

Arnon Rosenthal and César A. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proc. SIGMOD*, 1990.

Manfred Schmidt-Schaubß and Gert Smolka. Attributive concept descriptions with complements. *Artif. Intell.*, 48(1):1–26, 1991.

Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., sixth edition, 2010.

P. M. Stocker, P. M. D. Gray, and M. P. Atkinson, editors. *Databases – Role & Structure: An Advanced Course*. Cambridge University Press, 1984.

Michael Stonebraker. Technical perspective – one size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.

Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In *Proc. VLDB*, 2005.

Dan Suciu. Bounded fixpoints for complex objects. *Theor. Comput. Sci.*, 176(1–2):283–328, 1997.

V. Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *Proc. ICDT*, 1992.

Zahir Tari, John Stokes, and Stefano Spaccapietra. Object normal forms and dependency constraints for object-oriented schemata. *ACM Trans. Database Syst.*, 22(4):513–569, 1997.

Boris A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *American Mathematical Society Translations Series 2*, 23:1–5, 1963.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

Moshe Y. Vardi. Querying logical databases. In *Proc. PODS*, 1985.

Victor Vianu. Databases and finite-model theory. In *Proc. Descriptive Complexity and Finite Models*, 1996.

W3C. XML path language (XPath). http://www.w3.org/TR/xpath/, November 1999.

W3C. XML path language (XPath) 2.0. http://www.w3.org/TR/xpath20/, January 2007a.

W3C. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, January 2007b.

Marianne Winslett. Moshe Vardi speaks out on the proof, the whole proof, and nothing but the proof. *SIGMOD Record*, 35(1):56–64, 2006.

WinterCorp. 2005 TopTen award winners. Technical report, WinterCorp, 2005.

L. Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3):495–505, 1996.

Carlo Zaniolo. A new normal form for the design of relational database schemata. *ACM Trans. Database Syst.*, 7:489–499, 1982.