Andreas Meier
Michael Kaufmann

SQL & NoSQL Databases

Models, Languages, Consistency Options and Architectures for Big Data Management

Recommended in Germany

Springer Vieweg

SQL & NoSQL Databases

Andreas Meier · Michael Kaufmann

SQL & NoSQL Databases

Models, Languages, Consistency Options and Architectures for Big Data Management

Springer Vieweg

Andreas Meier
Department für Informatik
Universität Fribourg
Fribourg, Switzerland

Michael Kaufmann
Departement für Informatik
Hochschule Luzern
Rotkreuz, Switzerland

Translated from German by Anja Kreutel.

ISBN 978-3-658-24548-1 ISBN 978-3-658-24549-8 (eBook)
https://doi.org/10.1007/978-3-658-24549-8

Library of Congress Control Number: 2019935851

Springer Vieweg

This Springer Vieweg imprint is published by the registered company Springer Fachmedien Wiesbaden GmbH part of Springer Nature
The registered company address is: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Foreword

The term "database" has long since become part of people's everyday vocabulary, for managers and clerks as well as students of most subjects. They use it to describe a logically organized collection of electronically stored data that can be directly searched and viewed. However, they are generally more than happy to leave the whys and hows of its inner workings to the experts.

Users of databases are rarely aware of the immaterial and concrete business values contained in any individual database. This applies as much to a car importer's spare parts inventory as to the IT solution containing all customer depots at a bank or the patient information system of a hospital. Yet failure of these systems, or even cumulative errors, can threaten the very existence of the respective company or institution. For that reason, it is important for a much larger audience than just the "database specialists" to be well-informed about what is going on. Anyone involved with databases should understand what these tools are effectively able to do and which conditions must be created and maintained for them to do so.

Probably the most important aspect concerning databases involves (a) the distinction between their administration and the data stored in them (user data) and (b) the economic magnitude of these two areas. Database administration consists of various technical and administrative factors, from computers, database systems, and additional storage to the experts setting up and maintaining all these components—the aforementioned database specialists. It is crucial to keep in mind that the administration is by far the smaller part of standard database operation, constituting only about a quarter of the entire efforts.

Most of the work and expenses concerning databases lie in gathering, maintaining, and utilizing the user data. This includes the labor costs for all employees who enter data into the database, revise it, retrieve information from the database, or create files using this information. In the above examples, this means warehouse employees, bank tellers, or hospital personnel in a wide variety of fields—usually for several years.

In order to be able to properly evaluate the importance of the tasks connected with data maintenance and utilization on the one hand and database administration on the other hand, it is vital to understand and internalize this difference in the effort required

for each of them. Database administration starts with the design of the database, which already touches on many specialized topics such as determining the consistency checks for data manipulation or regulating data redundancies, which are as undesirable on the logical level as they are essential on the storage level. The development of database solutions is always targeted at their later use, so ill-considered decisions in the development process may have a permanent impact on everyday operations. Finding ideal solutions, such as the golden mean between too strict and too flexible when determining consistency conditions, may require some experience. Unduly strict conditions will interfere with regular operations, while excessively lax rules will entail a need for repeated expensive data repairs.

To avoid such issues, it is invaluable that anyone concerned with database development and operation, whether in management or as a database specialist, gain systematic insight into this field of computer sciences. The table of contents gives an overview of the wide variety of topics covered in this book. The title already shows that, in addition to an in-depth explanation of the field of conventional databases (relational model, SQL), the book also provides highly educational information about current advancements and related fields, the keywords being “NoSQL” or “post-relational” and “Big Data.” I am confident that the newest edition of this book will, once again, be well received by both students and professionals—its authors are quite familiar with both groups.

Carl August Zehnder

Preface

It is remarkable how stable some concepts are in the field of databases. Information technology is generally known to be subject to rapid development, bringing forth new technologies at an unbelievable pace. However, this is only superficially the case. Many aspects of computer science do not essentially change at all. This includes not only the basics, such as the functional principles of universal computing machines, processors, compilers, operating systems, databases and information systems, and distributed systems, but also computer language technologies such as C, TCP/IP, or HTML, which are decades old but in many ways provide a stable fundament of the global, earth-spanning information system known as the World Wide Web. Likewise, the SQL language has been in use for over four decades and will remain so in the foreseeable future. The theory of relational database systems was initiated in the 1970s by Codd (relation model and normal forms), Chen (entity and relationship model) and Chamberlin and Boyce (SEQUEL). However, these technologies have a major impact on the practice of data management today. Especially, with the Big Data revolution and the widespread use of data science methods for decision support, relational databases, and the use of SQL for data analysis are actually becoming more important. Even though sophisticated statistics and machine learning are enhancing the possibilities for knowledge extraction from data, many if not most data analyses for decision support rely on descriptive statistics using SQL for grouped aggregation. In that sense, although SQL database technology is quite mature, it is more relevant today than ever.

Nevertheless, a lot has changed in the area of database systems lately over the years. Especially the developments in the Big Data ecosystem brought new technologies into the world of databases, to which we pay enough attention to. The nonrelational database technologies, which are finding more and more fields of application under the generic term NoSQL, differ not only superficially from the classical relational databases, but also in the underlying principles. Relational databases were developed in the twentieth century with the purpose of enabling tightly organized, operational forms of data management, which provided stability but limited flexibility. In contrast, the NoSQL database movement emerged in the beginning of the current century, focusing on horizontal

partitioning and schema flexibility, and with the goal of solving the Big Data problems of volume, variety, and velocity, especially in Web-scale data systems. This has far-reaching consequences and has led to a new approach in data management, which deviates significantly from the previous theories on the basic concept of databases: the way data is modeled, how data is queried and manipulated, how data consistency is handled, and the system architecture. This is why we compare these two worlds, SQL and NoSQL databases, from different perspectives in all chapters.

We have also launched a website called sql-nosql.org, where we share teaching and tutoring materials such as slides, tutorials for SQL and Cypher, case studies, a workbench for MySQL and Neo4j, so that language training can be done either with SQL or with Cypher, the graph-oriented query language of the NoSQL database Neo4j.

At this point, we would like to thank Anja Kreutel for her great effort and success in translating the eighth edition of the German textbook to English. We also thank Alexander Denzler and Marcel Wehrle for the development of the workbench for relational and graph-oriented databases. For the redesign of the graphics, we were able to win Thomas Riediker and we thank him for his tireless efforts. He has succeeded in giving the pictures a modern style and an individual touch. For the further development of the tutorials and case studies, which are available on the website sql-nosql.org, we thank the computer science students Andreas Waldis, Bettina Willi, Markus Ineichen, and Simon Studer for their contributions to the tutorial in Cypher and to the case study Travelblitz with OpenOffice Base and with Neo4J. For the feedback on the manuscript we thank Alexander Denzler, Daniel Fasel, Konrad Marfurt, and Thomas Olnhoff, for their willingness to contribute to the quality of our work with their hints. A big thank you goes to Sybille Thelen, Dorothea Glaunsinger, and Hermann Engesser of Springer, who have supported us with patience and expertise.

February 2019

Andreas Meier
Michael Kaufmann

Contents

List of Figures

Data Management

1

1.1 Information Systems and Databases

The evolution from an industrial to an information and knowledge society is represented by the assessment of information as a factor in production. The following characteristics distinguish *information* from material goods:

- **Representation**: Information is specified by data (signs, signals, messages, or language elements).
- **Processing**: Information can be transmitted, stored, categorized, found, or converted into other representation formats using algorithms and data structures (calculation rules).
- **Combination**: Information can be freely combined. The origin of individual parts cannot be traced. Manipulation is possible at any point.
- **Age**: Information is not subject to physical aging processes.
- **Original**: Information can be copied without limit and does not distinguish between original and copy.
- **Vagueness**: Information is unclear, i.e., often imprecise and of differing validities (quality).
- **Medium**: Information does not require a fixed medium and is, therefore, independent of location.

These properties clearly show that digital goods (information, software, multimedia, etc.), i.e., data, are vastly different from material goods in both handling and economic or legal evaluation. A good example is the loss in value that physical products often experience when they are used—the shared use of information, on the other hand, may increase its value. Another difference lies in the potentially high production costs

A. Meier and M. Kaufmann, *SQL & NoSQL Databases,*
https://doi.org/10.1007/978-3-658-24549-8_1

for material goods, while information can be multiplied easily and at significantly lower costs (with only computing power and a storage medium). This causes difficulties in determining property rights and ownership, even though digital watermarks and other privacy and security measures are available.

Considering *data as the basis of information as a production factor* in a company has significant consequences:

- **Basis for decision-making**: Data allows well-informed decisions, making it vital for all organizational functions.
- **Quality level**: Data can be available from different sources; information quality depends on the availability, correctness, and completeness of the data.
- **Need for investments**: Data gathering, storage, and processing cause work and expenses.
- **Degree of integration**: Fields and holders of duties within any organization are connected by informational relations, meaning that the fulfillment of said duties largely depends on the degree of data integration.

Once data is viewed as a factor in production, it has to be planned, governed, monitored, and controlled. This makes it necessary to see data management as a task for the executive level, inducing a major change within the company: In addition to the technical function of operating the information and communication infrastructure (production), planning and design of data flows (application portfolio) are crucial.

As shown in Fig. 1.1, an *information system* enables users to store and connect information interactively, to ask questions, and to obtain answers. Depending on the type of information system, acceptable questions may be limited. There are, however, open

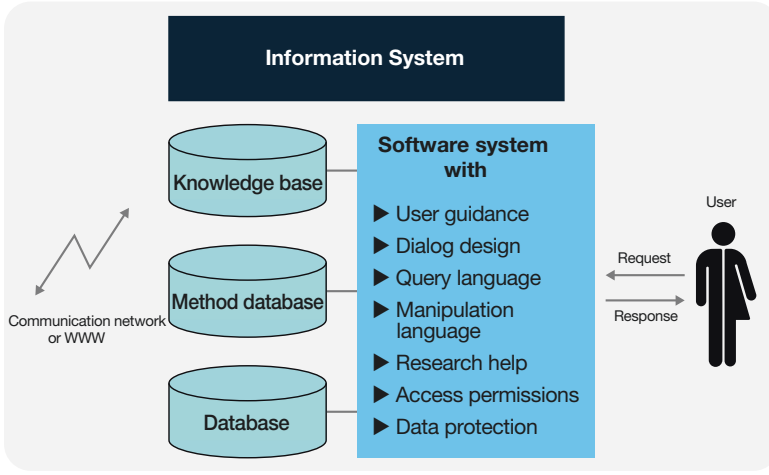


Fig. 1.1 Architecture and components of information systems

information systems and online platforms in the World Wide Web that use search engines to process arbitrary queries.

The computer-based information system in Fig. 1.1 is connected to a communication network/the World Wide Web in order to allow for online research and global information exchange in addition to company-specific analyses. Any information system of a certain size uses database technologies to avoid the necessity to redevelop data management and analysis every time it is used.

Database management systems are software for application-independently describing, storing, and querying data. All database management systems contain a storage and a management component. The storage component includes all data stored in an organized form plus their description. The management component contains a query and data manipulation language for evaluating and editing the data and information. This component does not only serve as the user interface, but also manages access and editing permissions for users.

SQL databases (SQL = Structured Query Language, cf., Sect. 1.2) are the most common in practical use. However, providing real-time web-based services referencing heterogeneous data sets is especially challenging (Sect. 1.3 on Big Data) and calls for new solutions such as NoSQL approaches (Sect. 1.4). When deciding whether to use relational or nonrelational technologies, the pros and cons have to be considered carefully—in some use cases, it may even be ideal to combine different technologies (cf., operating a web shop in Sect. 5.6). Depending on the database architecture of choice, data management within the company must be established and developed with the support of qualified experts (Sect. 1.5). References for further reading are listed in Sect. 1.6.

1.2 SQL Databases

1.2.1 Relational Model

One of the simplest and most intuitive ways to collect and present data is in a table. Most tabular data sets can be read and understood without additional explanations.

To collect information about employees, a table structure as shown in Fig. 1.2 can be used. The all-capitalized table name EMPLOYEE refers to the entire table, while the individual columns are given the desired attribute names as headers; in this example, the employee number "E#," the employee's name "Name," and their city of residence "City."

An *attribute* assigns a specific data value from a predefined value range called *domain* as a property to each entry in the table. In the EMPLOYEE table, the attribute E# allows individual employees to be uniquely identified, making it the *key* of the table. To mark key attributes more clearly, they are italicized in the table headers throughout this book. The attribute City is used to label the respective places of residence and the attribute Name for the names of the respective employees (Fig. 1.3).

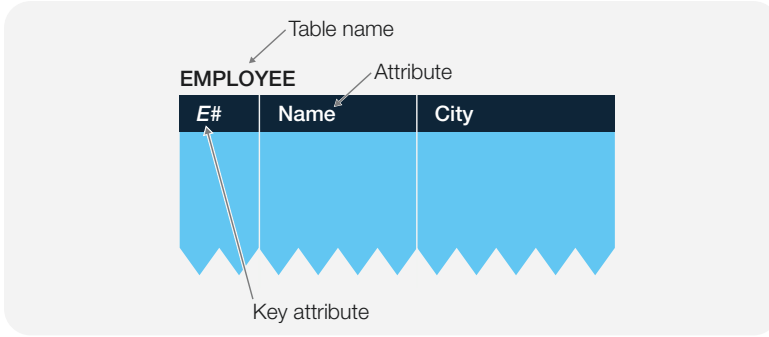


Fig. 1.2 Table structure for an EMPLOYEE table

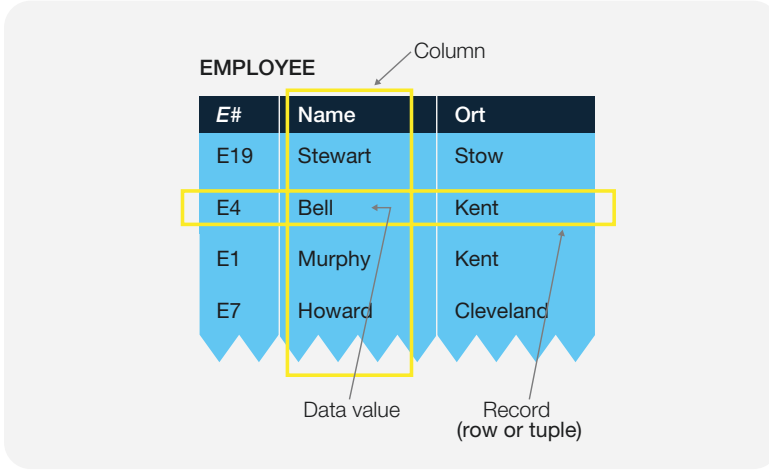


Fig. 1.3 EMPLOYEE table with manifestations

The required information of the employees can now easily be entered row by row. In the columns, values may appear more than once. In our example, Kent is listed as the place of residence of two employees. This is an important fact, telling us that both employee Murphy and employee Bell live in Kent. In our EMPLOYEE table, not only cities, but also employee names may exist multiple times. For this reason, the aforementioned key attribute E# is required to uniquely identify each employee in the table.

▶ **Identification key** The *identification key* or just *key* of a table is one attribute or a minimal combination of attributes whose values uniquely identify the records (called *rows* or *tuples*) within the table.

This short definition lets us infer two important *properties* of keys:

- **Uniqueness**: Each key value uniquely identifies one record within the table, i.e., different tuples must not have identical keys.
- **Minimality**: If the key is a combination of attributes, this combination must be minimal, i.e., no attribute can be removed from the combination without eliminating the unique identification.

The requirements of uniqueness and minimality fully characterize an identification key.

Instead of a natural attribute or a combination of natural attributes, an artificial attribute can be introduced into the table as key. The employee number E# in our example is an artificial attribute, as it is not a natural characteristic of the employees.

While we are hesitant to include artificial keys or numbers as identifying attributes, especially when the information in question is personal, natural keys often result in issues with uniqueness and/or privacy. For example, if a key is constructed from parts of the name and the date of birth, it may not necessarily be unique. Moreover, natural or intelligent keys divulge information about the respective person, potentially infringing on their privacy.

Due to these considerations, artificial keys should be defined *application-independent and without semantics* (meaning, informational value). As soon as any information can be deduced from the data values of a key, there is room for interpretation. Additionally, it is quite possible that the originally well-defined principle behind the key values changes or is lost over time.

▶ **Table definition** To summarize, a *table* or *relation* is a set of tuples presented in tabular form and meeting the following requirements:

- **Table name**: A table has a unique table name.
- **Attribute name**: All attribute names are unique within a table and label one specific column with the required property.
- **No column order**: The number of attributes is not set, and the order of the columns within the table does not matter.
- **No row order**: The number of tuples is not set, and the order of the rows within the table does not matter.
- **Identification key**: One attribute or a combination of attributes uniquely identifies the tuples within the table and is declared the identification key.

According to this definition, the *relational model* considers each table as a set of unordered tuples.

▶ **Relational model** The relational model represents both data and relationships between data as tables. Mathematically speaking, any relation R is simply a *subset of a*

Cartesian product of domains: $R \subseteq D_1 \times D_2 \times \ldots \times D_n$ with D_i as the domain of the i-th attribute/property. Any tuple r is, therefore, a set of specific data values or manifestations, $r = (d_1, d_2, \ldots, d_n)$. Please note that this definition means that any tuple may only exist once within any table, i.e., $R = \{r_1, r_2, \ldots, r_m\}$.

The relational model is based on the work of Edgar Frank Codd from the early 1970s. This was the foundation for the first *relational database systems*, created in research facilities and supporting SQL or similar database languages. Today, their sophisticated successors are firmly established in many practical uses.

1.2.2 Structured Query Language (SQL)

As explained, the relational model presents information in tabular form, where each table is a set of tuples (or records) of the same type. Seeing all the data as sets makes it possible to offer *query and manipulation options based on sets*.

The result of a selective operation, for example, is a set, i.e., each search result is returned by the database management system as a table. If no tuples of the scanned table show the respective properties, the user gets a blank results table. Manipulation operations similarly target sets and affect an entire table or individual table sections.

The primary query and data manipulation language for tables is called *Structured Query Language*, usually shortened to SQL (see Fig. 1.4). It was standardized by ANSI (American National Standards Institute) and ISO (International Organization for Standardization)[1].

SQL is a descriptive language, as the statements describe the desired result instead of the necessary computing steps. SQL queries follow a basic pattern as illustrated by the query in Fig. 1.4:

"SELECT the attribute Name FROM the EMPLOYEE table WHERE the city is Kent."

A SELECT-FROM-WHERE query can apply to one or several tables and always generates a table as a result. In our example, the query would yield a results table with the names Bell and Murphy, as desired.

The set-based method offers users a major advantage, since a single SQL query can trigger multiple actions within the database management system. It is not necessary for users to program all searches themselves.

Relational query and data manipulation languages are descriptive. Users get the desired results by merely setting the requested properties in the SELECT expression. They do not have to provide the procedure for computing the required records.

[1]ANSI is the national standards organization of the US. The national standardization organizations are part of ISO.

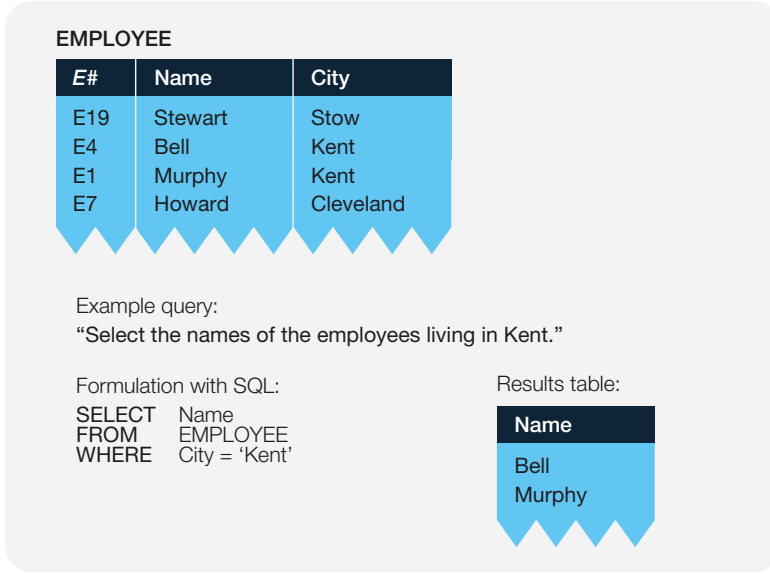


Fig. 1.4 Formulating a query in SQL

The database management system takes on this task, processes the query or manipulation with its own search and access methods, and generates the results table.

With procedural database languages, on the other hand, the methods for retrieving the requested information must be programmed by the user. In this case, each query yields only one record, not a set of tuples.

With its descriptive query formula, SQL requires only the specification of the desired selection conditions in the WHERE clause, while procedural languages require the user to specify an algorithm for finding the individual records. As an example, let us take a look at a query language for hierarchical databases (see Fig. 1.5): For our initial operation, we use GET_FIRST to search for the first record that meets our search criteria. Next, we access all other corresponding records individually with the command GET_NEXT until we reach the end of the file or a new hierarchy level within the database.

Overall, we can conclude that procedural database management languages use record-based or navigating commands to manipulate collections of data, requiring some experience and knowledge of the database's inner structure from the users. Occasional users basically cannot independently access and use the contents of a database. Unlike procedural languages, relational query and manipulation languages do not require the specification of access paths, processing procedures, or navigational routes, which significantly reduces the development effort for database utilization.

If database queries and analyses are to be done by company departments and end users instead of IT, the descriptive approach is extremely useful. Research on descriptive

Natural language:

"Select the names of the employees living in Kent."

Descriptive language:

```
SELECT   Name
FROM     EMPLOYEE
WHERE    City = 'Kent'
```

Procedural language:

```
get first EMPLOYEE
   search argument (City = 'Kent')
while status = 0 do
begin
   print (Name)
   get next EMPLOYEE
      search argument (City = 'Kent')
end
```

Fig. 1.5 The difference between descriptive and procedural languages

database interfaces has shown that *even occasional users have a high probability of successfully executing* the desired analyses using descriptive language elements. Figure 1.5 also illustrates the similarities between SQL and natural language. In fact, there are modern relational database management systems that can be accessed with natural language.

1.2.3 Relational Database Management System

Databases are used in the development and operation of information systems in order to store data centrally, permanently, and in a structured manner.

As shown in Fig. 1.6, relational database management systems are integrated systems for the consistent management of tables. They offer service functionalities and the descriptive language SQL for data description, selection, and manipulation.

Every relational database management system consists of a storage and a management component. The storage component stores both data and the relationships between pieces of information in tables. In addition to tables with user data from various applications, it contains the predefined system tables necessary for database operation. These contain descriptive information and can be queried but not manipulated by users.

The management component's most important part is the relational data definition, selection, and manipulation language SQL. This component also contains service functions for data restoration after errors, for data protection, and for backup.

Relational database management systems are common bases for businesses' information systems and can be defined as follows:

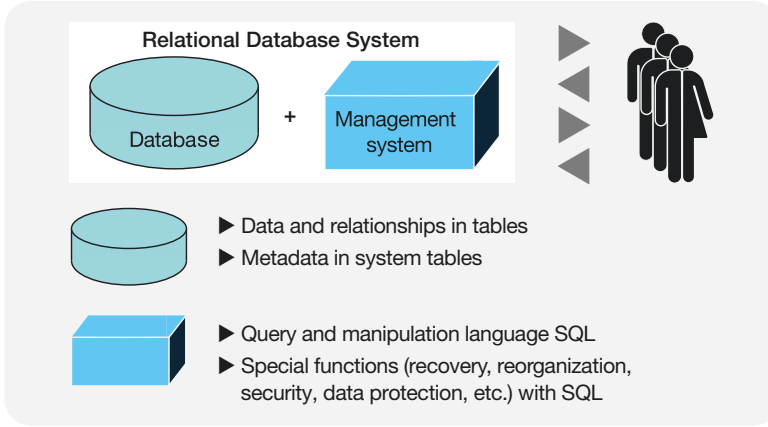


Fig. 1.6 Basic structure of a relational database management system

▶ **Relational database management system** *Relational database management systems* (RDBMS) have the following properties:

- **Model**: The database model follows the relational model, i.e., all data and data relations are represented in tables. Dependencies between attribute values of tuples or multiple instances of data can be discovered (cf., normal forms in Sect. 2.3.1).
- **Schema**: The definitions of tables and attributes are stored in the relational database schema. The schema further contains the definition of the identification keys and rules for integrity assurance.
- **Language**: The database system includes SQL for data definition, selection, and manipulation. The language component is descriptive and facilitates analyses and programming tasks for users.
- **Architecture**: The system ensures extensive *data independence*, i.e., data and applications are mostly segregated. This independence is reached by separating the actual storage component from the user side using the management component. Ideally, physical changes to relational databases are possible without the need to adjust related applications.
- **Multi-user operation**: The system supports multi-user operation (Sect. 4.1), i.e., several users can query or manipulate the same database at the same time. The RDBMS ensures that parallel transactions in one database do not interfere with each other or, worse, with the correctness of data (Sect. 4.2).
- **Consistency assurance**: The database management system provides tools for ensuring data integrity, i.e., the correct and uncompromised storage of data.
- **Data security and data protection**: The database management system provides mechanisms to protect data from destruction, loss, or unauthorized access (Sect. 3.8).

NoSQL database management systems meet these criteria only partially (Sect. 1.4.3 and Chaps. 4 and 7). For this reason, most corporations, organizations, and especially SMEs (small and medium enterprises) rely heavily on relational database management systems. However, for spread-out web applications or applications handling Big Data, relational database technology must be augmented with NoSQL technology in order to ensure uninterrupted global access to these services.

1.3 Big Data

The term Big Data is used to label large volumes of data that push the limits of conventional software. This data is usually unstructured (Sect. 5.1) and may originate from a wide variety of sources: social media postings, e-mails, electronic archives with multimedia content, search engine queries, document repositories of content management systems, sensor data of various kinds, rate developments at stock exchanges, traffic flow data and satellite images, smart meters in household appliances, order, purchase, and payment processes in online stores, e-health applications, monitoring systems, etc.

There is no binding definition for Big Data yet, but most data specialists will agree on three v's: *volume* (extensive amounts of data), *variety* (multiple formats, structured, semi-structured, and unstructured data, Fig. 1.7), and *velocity* (high-speed and real-time processing). Gartner Group's IT glossary offers the following definition:

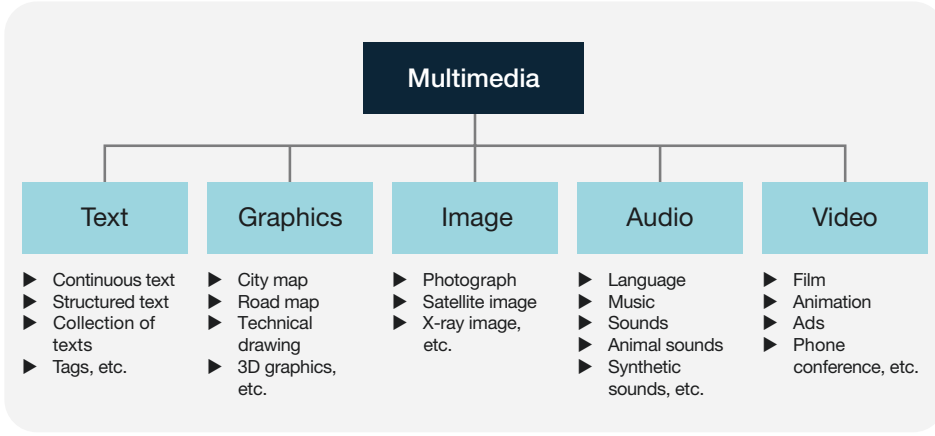


Fig. 1.7 Variety of sources for Big Data

Big Data
'*Big data is high-volume, high-velocity and high-variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making.*'[2]

With this definition, Gartner Group positions Big Data as *information assets* for companies. It is, indeed, vital for companies and organizations to generate decision-relevant knowledge in order to survive. In addition to internal information systems, they increasingly utilize the numerous resources available online to better anticipate economic, ecologic, and social developments on the markets.

Big Data is a challenge faced by not only for-profit companies in digital markets, but also governments, public authorities, NGOs (nongovernmental organizations), and NPOs (nonprofit organizations).

A good example are programs to create smart cities or ubiquitous cities, i.e., to use Big Data technologies in cities and urban agglomerations for sustainable development of social and ecologic aspects of human living spaces. They include projects facilitating mobility, the use of intelligent systems for water and energy supply, the promotion of social networks, expansion of political participation, encouragement of entrepreneurship, protection of the environment, and an increase of security and quality of life.

All use of Big Data applications requires successful management of the three v's mentioned above:

- **Volume**: There are massive amounts of data involved, ranging from tera to zettabytes (megabyte = 10^6 bytes, gigabyte = 10^9 bytes, terabyte = 10^{12} bytes, petabyte = 10^{15} bytes, exabyte = 10^{18} bytes, zettabyte = 10^{21} bytes).
- **Variety**: Big Data involves storing structured, semi-structured, and unstructured multimedia data (text, graphics, images, audio, and video; Fig. 1.7).
- **Velocity**: Applications must be able to process and analyze data streams (Sect. 5.1) in real-time as the data is gathered.
  As in the Gartner Group's definition, Big Data can be considered an information asset, which is why sometimes another V is added:
- **Value**: Big Data applications are meant to increase the enterprise value, so investments in personnel and technical infrastructure are made where they will bring leverage or added value can be generated.
  There are numerous open source solutions for NoSQL databases, and the technologies do not require expensive hardware, while they also offer good scalability. Specialized personnel, however, is lacking, since the data scientist profession (Sect. 1.5) is only

[2]Gartner Group, IT glossary—big data; http://www.gartner.com/it-glossary/big-data/, retrieved February 11, 2015

just emerging, and professional education in this sector is still in its pilot phase or only under discussion.

To complete our consideration of the concept of Big Data we will look at another V:

- **Veracity**: Since much data is vague or inaccurate, specific algorithms evaluating the validity and assessing result quality are needed. Large amounts of data do not automatically mean better analyses.

  Veracity is an important factor in Big Data, where the available data is of variable quality, which has to be taken into consideration in analyses. Aside from statistical methods, there are fuzzy methods of soft computing which assign a truth value between 0 (false) and 1(true) to any result or statement (fuzzy databases in Sect. 6.8).

1.4 NoSQL Databases

1.4.1 Graph-based Model

NoSQL databases support various database models (Sect. 1.4.3 and Fig. 1.11). We picked out graph databases as an example to look at and discuss their characteristics.

▶ **Property graph** *Property graphs* consists of *nodes* (concepts, objects) and *directed edges* (relationships) connecting the nodes. Both nodes and edges are given a *label* and can have *properties*. Properties are given as attribute-value pairs following the pattern (*attribute*: value) with the names of attributes and the respective values.

A graph abstractly presents the nodes and edges with their properties. Figure 1.8 shows part of a movie collection as an example. It contains the nodes MOVIE with attributes *Title* and *Year* (of release), GENRE with the respective *Type* (e.g., crime, mystery, comedy, drama, thriller, Western, science fiction, documentary, etc.), ACTOR with *Name* and *Year of Birth*, and DIRECTOR with *Name* and *Nationality*.

The example uses three directed edges: The edge ACTED_IN shows which artist from the ACTOR node starred in which film from the MOVIE node. This edge also has a property, the *Role* of the actor in the movie. The other two edges, HAS and DIRECTED_BY, go from the MOVIE node to the GENRE and DIRECTOR node, respectively.

In the manifestation level, i.e., the graph database, the property graph contains the concrete values (Fig. 1.9 in Sect. 1.4.2).

The property graph model for databases is formally based on graph theory. Depending on their maturity, relevant software products may offer algorithms to calculate the following traits:

- **Connectivity**: A graph is connected when every node in the graph is connected to every other node by at least one path.
- **Shortest path**: The shortest path between two nodes of a graph is the one with the least edges.

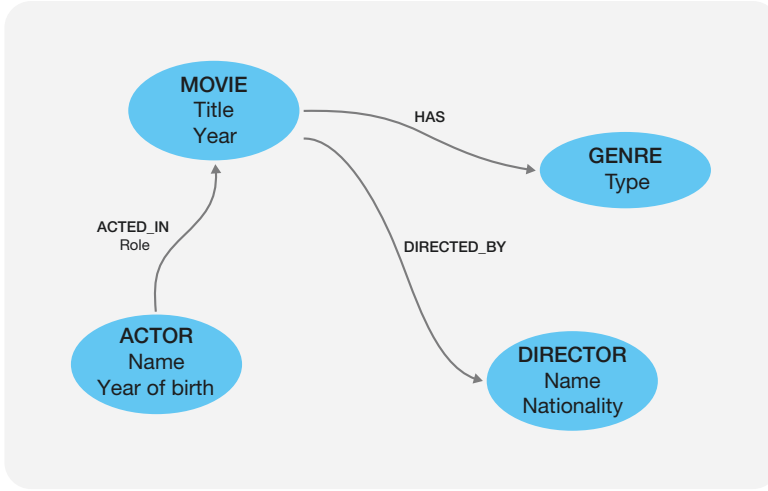


Fig. 1.8 Section of a property graph on movies

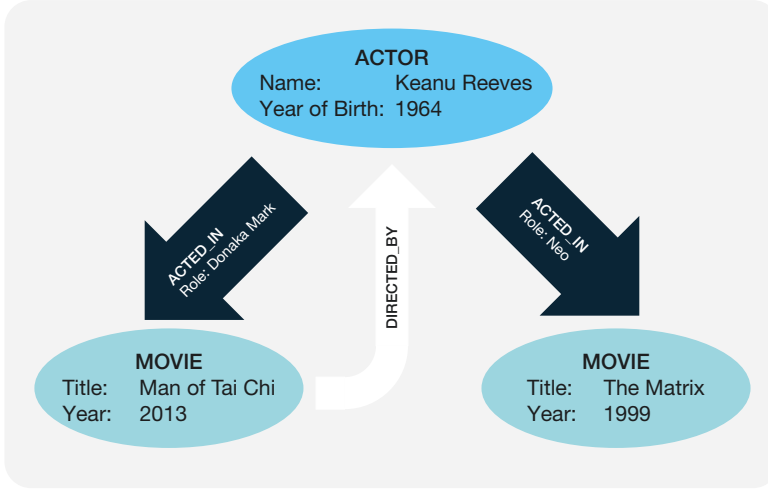


Fig. 1.9 Section of a graph database on movies

- **Nearest neighbor**: In graphs with weighted edges (e.g., by distance or time in a transport network), the nearest neighbors of a node can be determined by finding the minimal intervals (shortest route in terms of distance or time).
- **Matching**: Matching in graph theory means finding a set of edges that have no common nodes.

These graph characteristics are significant in many kinds of applications. Finding the shortest path or the nearest neighbor, for example, is of great importance in calculating travel or transport routes. The algorithms listed can also sort and analyze relationships in social networks by path length (Sect. 2.4).

1.4.2 Graph Query Language Cypher

Cypher is a declarative query language for extracting patterns from graph databases. Users define their query by specifying nodes and edges. The database management system then calculates all patterns meeting the criteria by analyzing the possible paths (connections between nodes via edges). In other words, the user declares the structure of the desired pattern, and the database management system's algorithms traverse all necessary connections (paths) and assemble the results.

As described in Sect. 1.4.1, the data model of a graph database consists of *nodes* (concepts, objects) and *directed edges* (relationships between nodes). In addition to their name, both nodes and edges can have a set of properties (see the Property Graph in Sect. 1.4.1). These properties are represented by attribute-value pairs.

Figure 1.9 shows a segment of a graph database on movies and actors. To keep things simple, only two types of node are shown: ACTOR and MOVIE. ACTOR nodes contain two attribute-value pairs, specifically (*Name*: FirstName LastName) and (*YearOfBirth*: Year).

The segment in Fig. 1.9 includes different types of edges: The ACTED_IN relationship represents which actors starred in which movies. Edges can also have properties if attribute-value pairs are added to them. For the ACTED_IN relationship, the respective roles of the actors in the movies are listed. For example, Keanu Reeves was cast as the hacker Neo in 'The Matrix.'

Nodes can be connected by multiple relationship edges. The movie 'Man of Tai Chi' and actor Keanu Reeves are linked not only by the actor's role (ACTED_IN), but also by the director position (DIRECTED_BY). The diagram therefore shows that Keanu Reeves both directed the movie 'Man of Tai Chi' and starred in it as Donaka Mark.

If we want to analyze this graph database on movies, we can use Cypher. It uses the following basic query elements:

- MATCH: Specification of nodes and edges, as well as declaration of search patterns.
- WHERE: Conditions for filtering results.
- RETURN: Specification of the desired search result, aggregated if necessary

For instance, the Cypher query for the year the movie 'The Matrix' was released would be:

```
MATCH   (m: Movie {Title: "The Matrix"})
RETURN  m.Year
```

The query sends out the variable m for the movie 'The Matrix' to return the movie's year of release by m.Year. In Cypher, parentheses always indicate nodes, i.e., (m: Movie) declares the control variable m for the MOVIE node. In addition to control variables, individual attribute-value pairs can be included in curly brackets. Since we are specifically interested in the movie 'The Matrix', we can add {Title: "The Matrix"} to the node (m: Movie).

Queries regarding the relationships within the graph database are a bit more complicated. Relationships between two arbitrary nodes (a) and (b) are expressed in Cypher by the arrow symbol "- >", i.e., the path from (a) to (b) is declared as "(a)- >(b)". If the specific relationship between (a) and (b) is of importance, the edge [r] can be inserted in the middle of the arrow. The square brackets represent edges, and r is our variable for relationships.

Now, if we want to find out who played Neo in 'The Matrix', we use the following query to analyze the ACTED_IN path between ACTOR and MOVIE:

```
MATCH    (a: Actor)-[: Acted_In {Role: "Neo"}]- >
         (: Movie {Title: "The Matrix"}])
RETURN   a.Name
```

Cypher will return the result Keanu Reeves.

For a list of movie titles (m), actor names (a), and respective roles (r), the query would have to be:

```
MATCH    (a: Actor)-[r: Acted_In] - > (m: Movie)
RETURN   m.Title, a.Name, r.Role
```

Since our example graph database only contains one actor and two movies, the result would be the movie 'Man of Tai Chi' with actor Keanu Reeves in the role of Donaka Mark and the movie 'The Matrix' with Keanu Reeves as Neo.

In real life, however, such a graph database of actors, movies, and roles has countless entries. A manageable query would, therefore, have to remain limited, e.g., to actor Keanu Reeves, and would then look like this:

```
MATCH    (a: Actor)-[r: Acted_In] - >(m: Movie)
WHERE    (a.Name = "Keanu Reeves")
RETURN   m.Title, a.Name, r.Role
```

Similar to SQL, Cypher uses declarative queries where the user specifies the desired properties of the result pattern (Cypher) or results table (SQL), and the respective database management system then calculates the results. However, analyzing relationship networks, using recursive search strategies, or analyzing graph properties are hardly possible with SQL.

1.4.3 NoSQL Database Management System

Before Ted Codd's introduction of the relational model, nonrelational databases such as hierarchical or network-like databases existed. After the development of relational database management systems, nonrelational models were still used in technical or scientific applications. For instance, running CAD (computer-aided design) systems for structural or machine components on relational technology is rather difficult. Splitting technical objects across a multitude of tables proved problematic, as geometric, topological, and graphical manipulations all had to be executed in real time.

The advent of the internet and numerous web-based applications has provided quite a boost to the relevance of nonrelational data concepts versus relational ones, as managing Big Data applications with relational database technology is difficult to impossible.

While 'nonrelational' would be a better description than NoSQL, the latter has become established with database researchers and providers on the market over the last few years.

▶ **NoSQL** The term NoSQL is now used for any *nonrelational data management approaches* meeting two criteria:

- Firstly: The data is not stored in tables.
- Secondly: The database language is not SQL.

NoSQL is also sometimes interpreted as 'Not only SQL' to express that other technologies besides relational data technology are used in massively distributed web applications. NoSQL technologies are especially necessary if the web service requires high availability. Section 5.6 discusses the example of an online shop that uses various NoSQL technologies in addition to a relational database (Fig. 5.13).

The basic structure of an NoSQL database management system is shown in Fig. 1.10. NoSQL database management systems mostly use a massively distributed storage architecture. The actual data is stored in key-value pairs, columns or column families, document stores, or graphs (Fig. 1.11 and Chap. 7). In order to ensure high availability and avoid outages in NoSQL database systems, various redundancy concepts (cf., "consistent hashing" in Sect. 5.2.3) are supported.

The massively distributed and replicated architecture also enables parallel analyses ("MapReduce" in Sect. 5.4). Especially analyses of large volume of data or the search for specific information can be significantly accelerated with distributed computing processes. In the map/reduce method, subtasks are delegated to various computer nodes and simple key-value pairs are extracted (map), then the partial results are aggregated and returned (reduce).

There are also multiple consistency models or massively distributed computing networks (Sect. 4.3). Strong consistency means that the NoSQL database management system ensures full consistency at all times. Systems with weak consistency tolerate that changes will be copied to replicated nodes with a delay, resulting in

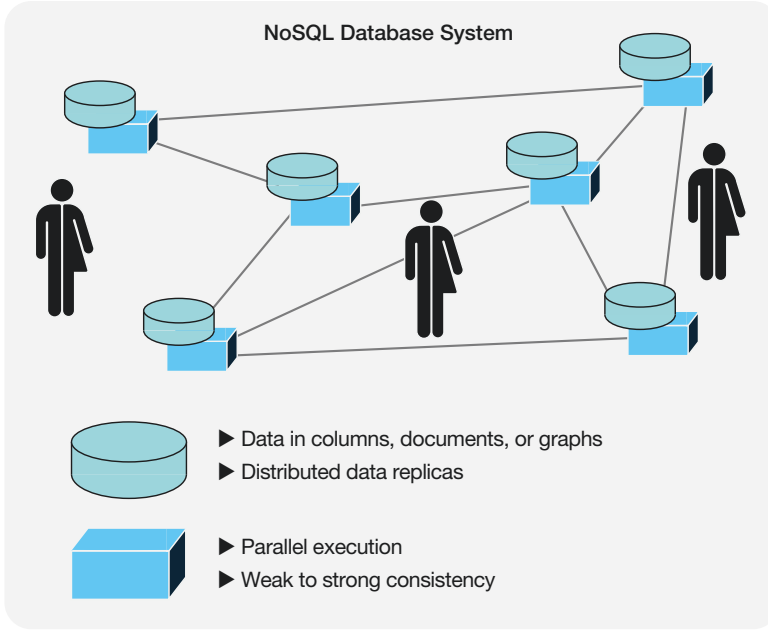


Fig. 1.10 Basic structure of a NoSQL database management system

temporary inconsistencies. Further differentiation is possible, e.g., consistency by quorum (Sect. 4.3.2).

The following definition of NoSQL databases is guided by the web-based NoSQL Archive[3]:

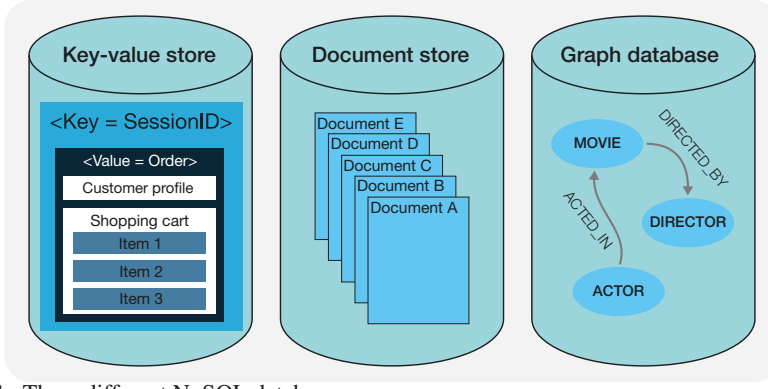


Fig. 1.11 Three different NoSQL databases

[3]NoSQL Archive; http://nosql-database.org/, retrieved February 17, 2015

▶ **NoSQL database system** Web-based storage systems are considered *NoSQL database systems* if they meet the following requirements:

- **Model**: The underlying database model is not relational.
- **At least three Vs**: The database system includes a large amount of data (volume), flexible data structures (variety), and real-time processing (velocity).
- **Schema**: The database management system is not bound by a fixed database schema.
- **Architecture**: The database architecture supports massively distributed web applications and horizontal scaling.
- **Replication**: The database management system supports data replication.
- **Consistency assurance**: According to the CAP theorem (Sect. 4.3.1), consistency may be ensured with a delay to prioritize high availability and partition tolerance.

The researchers and operators of the NoSQL Archive list more than 225 NoSQL database products on their website, most of them open source. However, the multitude of solutions indicates that the market for NoSQL products is not yet completely secure. Moreover, implementation of suitable NoSQL technologies requires specialists who know not only the underlying concepts, but also the various architectural approaches and tools.

Figure 1.11 shows three different NoSQL database management systems.

Key-value stores (Sect. 7.2) are the simplest version. Data is stored as an identification key<key = "key">and a list of values<value = "value 1", "value 2", …>. A good example is an online store with session management and shopping basket. The session ID is the identification key; the individual items from the basket are stored as values in addition to the customer profile.

In *document stores* (Sect. 7.4), records are managed as documents within the NoSQL database. These documents are structured text files, e.g., in JSON or XML format, which can be searched for by a unique key or attributes within the documents. Unlike key-value stores, documents have some structure; however, it is schema free, i.e., the structures of individual records (documents) can vary.

The third example revisits the *graph database* on movies and actors discussed in the previous sections (for more details on graph databases, see also Sect. 7.6).

1.5 Organization of Data Management

Many companies and organizations view their data as a vital resource, increasingly joining in public information gathering in addition to maintaining their own data. The continuous global increase and growth of information providers and their 24/7 services reinforce the importance of web-based data pools.

The necessity for current information based in the real world has a direct impact on the conception of the field of IT. In many places, specific positions for data management have been created for a more targeted approach to data-related tasks and obligations. Pro-active data management deals both strategically with information gathering and utilization and operatively with the efficient provision and analysis of current and consistent data.

Development and operation of data management incur high costs, while the return is initially hard to measure. Flexible data architecture, noncontradictory and easy-to-understand data description, clean and consistent databases, effective security concepts, current information readiness, and other factors involved are hard to assess and include in profitability considerations. Only the realization of the data's importance and longevity makes the necessary investments worthwhile for the company.

For better comprehension of the term data management, we will look at the four subfields: data architecture, data governance, data technology, and data utilization. Figure 1.12 illustrates the objectives and tools of these four fields within data management.

Employees in *data architecture* analyze, categorize, and structure the company data with a sophisticated methodology. In addition to the assessment of data and information requirements, the major data classes and their relationships with each other must be

	Tasks	Tools
Data architecture	▶ Creation and maintenance of the company-wide data architecture ▶ Definition of data protection rules	▶ Data analysis and design methodology ▶ Tools for computer-based information modeling
Data administration	▶ Management of data and methods using standardization guidelines and international standards ▶ Consultation for developers and end users	▶ Data dictionary systems ▶ Tools for cross-reference and usage lists
Data technology	▶ Installation, reorganization, and safe-keeping of data content ▶ Definition of the distribution concept incl. replication ▶ Disaster prevention and recovery	▶ Various database system services ▶ Tools for performance optimization ▶ Monitoring systems ▶ Tools for recovery/restart
Data utilization	▶ Data analysis and interpretation ▶ Knowledge generation ▶ Formulation of predictions ▶ Pattern detection	▶ Analysis tools ▶ Report generators ▶ Data mining tools ▶ Visualization methods for multi-dimensional data

Fig. 1.12 The four cornerstones of data management

documented in data models of varying specificity. These models, created from abstraction of reality and matched to each other, form the foundation of the data architecture.

Data governance aims for a unified coverage of data descriptions and formats as well as the respective responsibilities in order to ensure a cross-application use of the long-lived company data. Today's tendency towards decentralized data storage on intelligent workplace computers or distributed department machines is leading to a growing responsibility of data governance experts for maintaining data and assigning permissions.

Data technology specialists install, monitor, and reorganize databases and are in charge of their multilayer security. Their field, also known as database technology or database administration, further includes technology management and the need for the integration of new extensions and constant updates and improvements of existing tools and methods.

The fourth column of data management, *data utilization*, enables the actual, profitable application of business data. A specialized team of data scientists (see job profile below) conducts business analytics, providing and reporting on data analyses to management. They also support individual departments, e.g., marketing, sales, customer service, etc., in generating specific relevant insights from Big Data.

Based on the characterization of data-related tasks and obligations, data management can be defined as:

▶ **Data management** *Data management* includes all *operational*, *organizational*, and *technical aspects* of data architecture, data governance, and data technology that support company-wide *data storage*, *maintenance*, and *utilization,* as well as *business analytics*.

Over the past years, new specializations have evolved in the data management field, most importantly:

- **Data architects**: Data architects are in charge of a companies' entire data architecture. They decide where and how data has to be accessible in the respective business model and collaborate with database specialists on questions of data distribution, replication, or fragmentation.
- **Database specialists**: Database specialists are experts on database and system technology and manage the physical implementation of the data architecture. They decide which database management systems (SQL and/or NoSQL) to use for which application architecture components. Moreover, they are responsible for designing a distribution concept and for archiving, reorganizing, and restoring existing data.
- **Data scientists**: Data scientists are business analytics experts. They handle data analysis and interpretation, extracting previously unknown facts from data (knowledge generation) and providing prognoses for future business developments. Data scientists use methods and tools from data mining (pattern recognition), statistics, and visualization of multidimensional connections between data.

The conceptualization proposed above for both data management and the occupational profiles involved contains technical, organizational, and operational aspects. However, that does not mean that all roles within data architecture, data governance, data technology, and data utilization must be consolidated into one unit within the structure of a company or organization.

1.6 Further Reading

The wide range of literature on the subject of databases shows the importance of this field of IT. Some books describe not only relational, but also distributed, object-oriented, or knowledge-based database management systems. Well-known works include Connolly and Begg (2014), Hoffer et al. (2012), and Date (2004). The textbook by Ullman (1982) follows a rather theoretical approach, Silberschatz et al. (2010) is quite informative, and for a comprehensive overview, turn to Elmasri and Navathe (2015) or Garcia-Molina et al. (2014). Gardarin and Valduriez (1989) offer an introduction to relational database technology and knowledge bases.

German works of note in the field of databases include Lang and Lockemann (1995), Schlageter and Stucky (1983), Wedekind (1981), and Zehnder (2005). The textbooks by Saake et al. (2013), Kemper and Eickler (2013), and Vossen (2008) explore the foundations and extensions of database systems. Our definition of databases is based on the work of Claus and Schwill (2001).

As for Big Data, the market has been flooded with books over the recent years; however, most of them merely give a superficial description of the subject. Two short introductions by Celko (2014) and Sadalage and Fowler (2013) explain the terminology and present the most influential NoSQL database approaches. The work of Redmond and Wilson (2012) provides concrete descriptions of seven database management systems for a more in-depth technical understanding.

There are also some German publications on the topic of Big Data. The book by Edlich et al. (2011) offers an introduction to NoSQL database technologies and presents various products for key-value store, document store, column store, and graph databases, respectively. Freiknecht (2014) describes Hadoop, a popular framework for scalable and distributed systems, including its components for data storage (HBase) and data warehousing (Hive). The volume compiled by Fasel and Meier (2016) provides an overview over the development of Big Data in business environments—introducing the major NoSQL databases, presenting use cases, discussing legal aspects, and giving practical implementation advice.

For technical information on operational aspects of data management, we recommend Dippold et al. (2005). Biethahn et al. (2000) dedicate several chapters of their volume on data and development management to data architecture and governance. Heinrich and Lehner (2005) and Österle et al. (1991) touch on some facets of data management in their books on information management, while Meier's (1994) article defines the goals and tasks of data management from a practical perspective. Meier and Johner (1991) and Ortner et al. (1990) also handle some aspects of data governance.

References

Biethahn, J., Mucksch, H., Ruf, W.: Ganzheitliches Informationsmanagement (Bd. II: Daten- und Entwicklungsmanagement). Oldenbourg, München (2000)

Celko, J.: Joe Celko's Complete Guide to NoSQL—What Every SQL Professional Needs to Know About Nonrelational Databases. Morgan Kaufmann, Amsterdam (2014)

Claus, V., Schwill, A.: Duden Informatik. Ein Fachlexikon für Studium und Praxis. 3. Edition. Duden, Mannheim (2001)

Connolly, T., Begg, C.: Database Systems—A Practical Approach to Design, Implementation and Management. Addison-Wesley, Boston (2014)

Date, C.J.: An Introduction to Database Systems. Addison-Wesley, Boston (2004)

Dippold, R., Meier, A., Schnider, W., Schwinn, K.: Unternehmensweites Datenmanagement—Von der Datenbankadministration bis zum Informationsmanagement. Vieweg, Wiesbaden (2005)

Edlich, S., Friedland, A., Hampe, J., Brauer, B., Brückner, M.: NoSQL—Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken. Hanser, München (2011)

Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems. Addison-Wesley, Boston (2015)

Fasel, D., Meier, A. (eds.): Big Data—Grundlagen, Systeme und Nutzungspotenziale. Edition HMD. Springer, Wiesbaden (2016)

Freiknecht, J.: Big Data in der Praxis—Lösungen mit Hadoop, HBase und Hive—Daten speichern, aufbereiten und visualisieren. Hanser, München (2014)

Garcia-Molina, H., Ullman, J.D., Widom: Database Systems—The Complete Book. Pearson Education Limited, Harlow (2014)

Gardarin, G., Valduriez, P.: Relational Databases and Knowledge Bases. Addison Wesley, Mass. (1989)

Heinrich, L.J., Lehner, F.: Informationsmanagement—Planung, Überwachung und Steuerung der Informationsinfrastruktur. Oldenbourg, München (2005)

Hoffer, I.A., Prescott, M.B., Toppi, H.: Modern Database Management. Prentice Hall, Upper Saddle River (2012)

Kemper, A., Eickler, A.: Datenbanksysteme—Eine Einführung. Oldenbourg, München (2013)

Lang, S.M., Lockemann, P.C.: Datenbankeinsatz. Springer, Berlin (1995)

Meier, A.: Ziele und Aufgaben im Datenmanagement aus der Sicht des Praktikers. Wirtschaftsinformatik **36**(5), 455–464 (1994)

Meier, A., Johner, W.: Ziele und Pflichten der Datenadministration. Theorie und Praxis der Wirtschaftsinformatik **28**(161), 117–131 (1991)

Ortner, E., Rössner, J., Söllner, B.: Entwicklung und Verwaltung standardisierter Datenelemente. Informatik-Spektrum **13**(1), 17–30 (1990)

Österle, H., Brenner, W., Hilbers, K.: Unternehmensführung und Informationssystem—Der Ansatz des St. Galler Informationssystem-Managements. Teubner, Wiesbaden (1991)

Redmond, E., Wilson, J.R.: Seven Databases in Seven Weeks—A Guide to Modern Databases and The NoSQL Movement. The Pragmatic Bookshelf, Dallas (2012)

Saake G., Sattler K.-H., Heuer A.: Datenbanken—Konzepte und Sprachen. mitp, Frechen (2013)

Sadalage, P.J., Fowler, M.: NoSQL Distilled—A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, Upper Saddle River (2013)

Schlageter, G., Stucky, W.: Datenbanksysteme—Konzepte und Modelle. Teubner, Stuttgart (1983)

Silberschatz, A., Korth, H.F., Sudarshan, S.: Database Systems Concepts. McGraw-Hill, New York (2010)

Ullman, J.: Principles of Database Systems. Computer Science Press, Rockville (1982)

Vossen, G.: Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme. München, Oldenbourg (2008)
Wedekind, H.: Datenbanksysteme—Eine konstruktive Einführung in die Datenverarbeitung in Wirtschaft und Verwaltung. Spektrum Akademischer Verlag, Mannheim (1981)
Zehnder, C.A.: Informationssysteme und Datenbanken. vdf Hochschulverlag, Zürich (2005)

Data Modeling

2

2.1 From Data Analysis to Database

Data models provide a structured and formal description of the data and data relationships required for an information system. When data is needed for IT projects, such as the information about employees, departments, and projects in Fig. 2.1, the necessary data categories and their relationships with each other can be defined. The definition of those data categories, called entity sets, and the determination of relationship sets is at this point done without considering the kind of database management system (SQL or NoSQL) to be used for entering, storing, and maintaining the data later. This is to ensure that the data and data relationships will remain *stable from the users' perspective* throughout the development and expansion of information systems.

It takes three steps to set up a database for describing a section of the real world: data analysis, designing a conceptual data model (here: entity-relationship model), and converting it into a relational or nonrelational database schema.

The goal of data analysis (see point 1 in Fig. 2.1) is to find, in cooperation with the user, the data required for the information system and their relationships to each other including the quantity structure. This is vital for an early determination of the system boundaries. The requirement analysis is prepared in an iterative process, based on interviews, demand analyses, questionnaires, form compilations, etc. It contains at least a verbal task description with clearly formulated objectives and a *list of relevant pieces of information* (see the example in Fig. 2.1). The written description of data connections can be complemented by graphical illustrations or a summarizing example. It is imperative that the data analysis puts the facts necessary for the later development of a database in the language of the users.

Step 2 in Fig. 2.1 shows the conception of the *entity-relationship model*, which contains both the required entity sets and the relevant relationship sets. Our model depicts

A. Meier and M. Kaufmann, *SQL & NoSQL Databases,*
https://doi.org/10.1007/978-3-658-24549-8_2

the entity sets as rectangles and the relationship sets as rhombi. Based on the data analysis from step 1, the main entity sets are DEPARTMENT, EMPLOYEE, and PROJECT[1]. To record which departments the employees are working in and which projects they are part of, the relationship sets SUBORDINATE and INVOLVED are established and graphically connected to the respective entity sets. The entity-relationship model, therefore, allows for the structuring and graphic representation of the facts gathered during data analysis. However, it should be noted that the identification of entity and relationship sets, as well as the definition of the relevant attributes is not always a simple, clear-cut process. Rather, this design step requires quite some experience and practice from the data architect.

Next, the entity-relationship model is converted into a *relational database schema* (Fig. 2.1, 3a) or a *graph-oriented database schema* (Fig. 2.1, 3b). A database schema is the formal description of the database objects using either tables or nodes and edges.

Since relational database management systems allow only tables as objects, *both the entity and the relationship sets must be expressed as tables*. For this reason, there is one entity set table each for the entity sets DEPARTMENT, EMPLOYEE, and PROJECT in Fig. 2.1, step 3a. In order to represent the relationships in tables as well, separate tables have to be defined for each relationship set. In our example, this results in the tables SUBORDINATE and INVOLVED. Such relationship set tables always contain the keys of the entity sets affected by the relationship as foreign keys and potentially additional attributes of the relationship.

In step 3b of Fig. 2.1, we see the depiction of an equivalent graph database. Each entity set corresponds to a node in the graph, so we have the nodes DEPARTMENT, EMPLOYEE, and PROJECT. The relationship sets SUBORDINATE and INVOLVED from the entity-relationship model are converted into edges in the graph-based model. The relationship set SUBORDINATE becomes a directed edge from the DEPARTMENT node to the EMPLOYEE node and is named HAS_AS_SUBORDINATE. Similarly, a directed edge with the name IS_INVOLVED is drawn from the EMPLOYEE node to the PROJECT node.

This is only a rough sketch of the process of data analysis, development of an entity-relationship model, and definition of a relational or graph-based database schema. The core insight is that a database design should be developed based on an entity-relationship model. This allows for the gathering and discussion of data modeling factors with the users, independently of any specific database system. Only in the next design step is the most suitable database schema determined and mapped out. Both for relational and for graph-oriented databases, there are clearly defined mapping rules (Sects. 2.3.2 and 2.4.2, respectively).

[1]The names of entity and relationship sets are spelled in capital letters, analogous to table, node, and edge names.

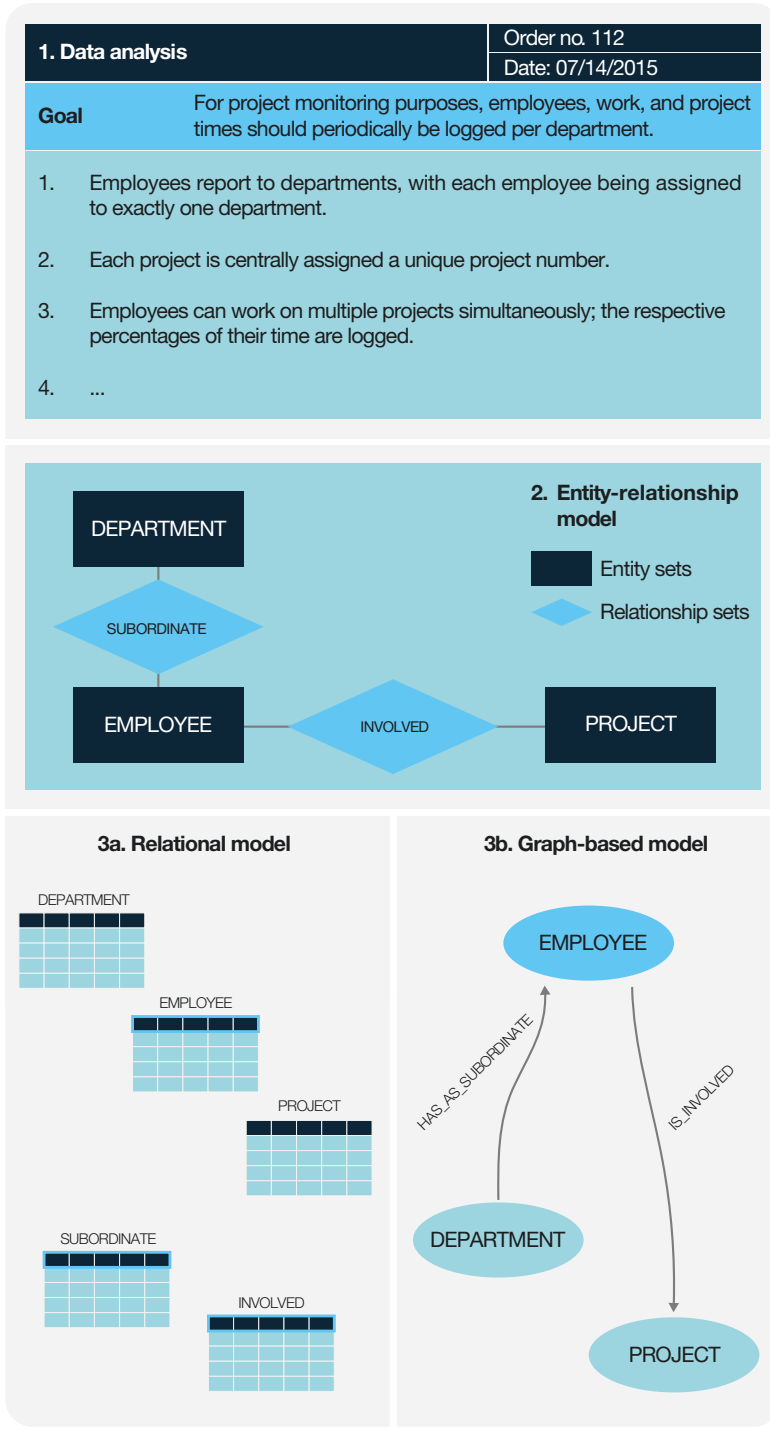


Fig. 2.1 The three steps necessary for data modeling

Section 2.2 explores the entity-relationship model in more detail, including the methods of generalization and aggregation. Section 2.3 discusses modeling aspects for relational databases, and Sect. 2.4 for graph databases; both explain the respective mapping rules for entity and relationship sets, as well as generalization and aggregation. Section 2.5 illustrates the necessity to develop a ubiquitous data architecture within an organization. A formula for the analysis, modeling, and database steps is provided in Sect. 2.6, and a short literature review can be found in Sect. 2.7.

2.2 The Entity-Relationship Model

2.2.1 Entities and Relationships

An *entity* is a specific object in the real world or our imagination that is distinct from all others. This can be an individual, an item, an abstract concept, or an event. Entities of the same type are combined into *entity sets* and are further characterized by attributes. These attributes are property categories of the entity and/or the entity set, such as size, name, weight, etc.

For each entity set, an identification key, i.e., one attribute or a specific combination of attributes, is set as unique. In addition to uniqueness, it also has to meet the criterion of the minimal combination of attributes for identification keys as described in Sect. 1.2.1.

In Fig. 2.2, an individual employee is characterized as an entity by their concrete attributes. If, in the course of internal project monitoring, all employees are to be listed with their names and address data, an entity set EMPLOYEE is created. An artificial

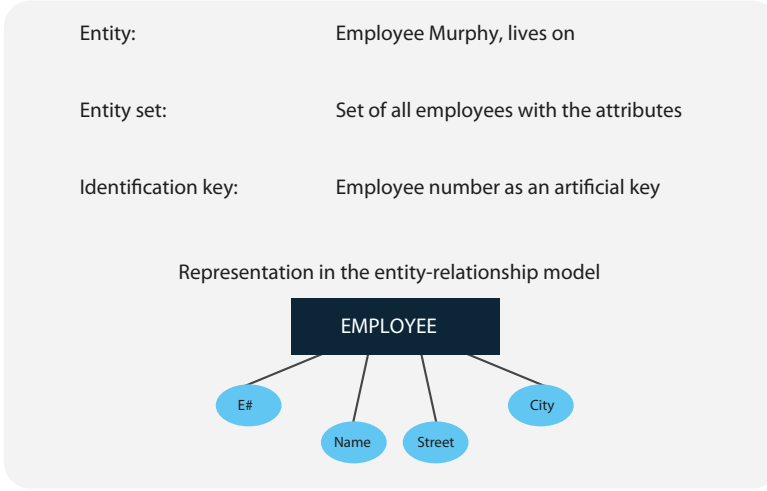


Fig. 2.2 EMPLOYEE entity set

employee number in addition to the attributes Name, Street, and City allows for the unique identification of the individual employees (entities) within the staff (entity set).

Besides the entity sets themselves, the *relationships* between them are of interest and can form sets of their own. Similar to entity sets, relationship sets can be characterized by attributes.

Figure 2.3 presents the statement "Employee Murphy does 70% of her work on project P17" as a concrete example of an employee-project relationship. The respective relationship set INVOLVED is to list all project participations of the employees. It contains a concatenated key constructed from the foreign keys employee number and project number. This combination of attributes ensures the unique identification of each project participation by an employee. Along with the concatenated key, the relationship set receives its own attribute named "Percentage" specifying the percentage of working hours that employees allot to each project they are involved in.

In general, relationships can be understood as associations in two directions: The relationship set INVOLVED can be interpreted from the perspective of the EMPLOYEE entity set as 'one employee can participate in multiple projects'; from the entity set PROJECT as 'one project is handled by multiple employees'.

2.2.2 Association Types

The *association* of an entity set ES_1 to another entity set ES_2 is the meaning of the relationship in that direction. As an example, the relationship DEPARTMENT_HEAD in Fig. 2.4 has two associations: On one hand, each department has one employee in the

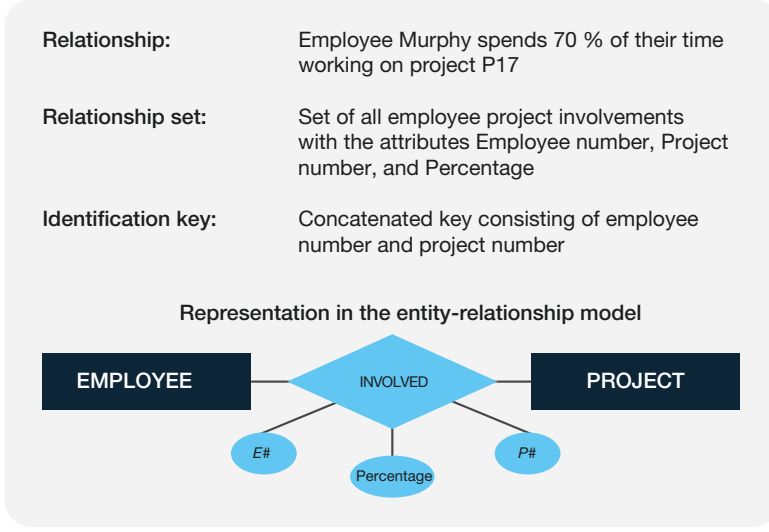


Fig. 2.3 INVOLVED relationship between employees and projects

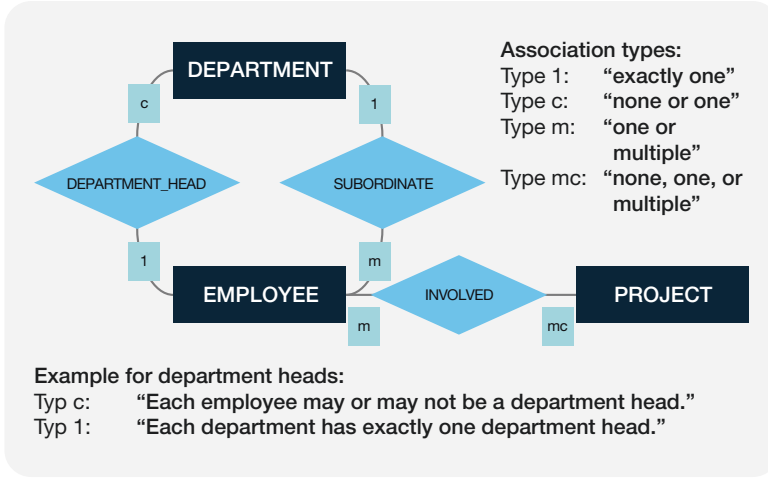


Fig. 2.4 Entity-relationship model with association types

role of department head, on the other hand, some employees could fill the role of department head for a specific department.

Each association from an entity set ES_1 to an entity set ES_2 can be weighted by an association type. The association type from ES_1 to ES_2 indicates how many entities of the associated entity set ES_2 can be assigned to a specific entity from ES_1[2]. The main distinction is between single, conditional, multiple, and multiple-conditional association types.

▶ **Unique association (type 1)** In unique, or type 1, associations, each entity from the entity set ES_1 is assigned *exactly one* entity from the entity set EM_2. For example, our data analysis showed that each employee is subordinate to exactly one department, i.e., matrix management is not permitted. The SUBORDINATE relationship from employees to departments in Fig. 2.4 is, therefore, a unique/type 1 association.

▶ **Conditional association (type c)** A type c association means that each entity from the entity set ES_1 is assigned *zero or one*, i.e., maximum one, entity from the entity set ES_2. The relationship is optional, so the association type is *conditional*. An example of a conditional association is the relationship DEPARTMENT_HEAD (Fig. 2.4), since not every employee can have the role of a department head.

[2]It is common in database literature to note the association type from ES_1 to ES_2 next to the associated entity set, i.e., ES_2.

▶ **Multiple association (type m)** In multiple, or type m, associations, each entity from the entity set ES_1 is assigned *one or more* entities from the entity set EM_2. This association type is often called *complex*, since one entity from ES_1 can be related to an arbitrary number of entities from ES_2. An example for the multiple association type in Fig. 2.4 is the INVOLVED relationship from projects to employees: Each project can involve multiple employees, but must be handled by at least one.

▶ **Multiple-conditional association (type mc)** Each entity from the entity set ES_1 is assigned *zero, one, or multiple* entities from the entity set ES_2. Multiple-conditional associations differ from multiple associations in that not every entity from ES_1 must have a relationship to any entity in ES_2. In analogy to that type, they are also called conditional-complex. We will exemplify this with the INVOLVED relationship in Fig. 2.4 as well, but this time from the employees' perspective: While not every employee has to participate in projects, there are some employees involved in multiple projects.

The association types provide information about the cardinality of the relationship. As we have seen, each relationship contains two association types. The *cardinality of a relationship* between the entity sets ES_1 and ES_2 is, therefore, a *pair of association types* of the form:

Cardinality: = (association type from ES_1 to ES_2, association type from ES_2 to ES_1)[3]

For example, the pair (mc,m) of association types between EMPLOYEE and PROJECT indicates that the INVOLVED relationship is (multiple-conditional, multiple).

Figure 2.5 shows all 16 possible combinations of association types. The first quadrant contains four options of unique-unique relationships (case B1 in Fig. 2.5). They are characterized by the cardinalities (1,1), (1,c), (c,1), and (c,c). For case B2, the unique-complex relationships, also called *hierarchical relationships*, there are eight possible combinations. The complex-complex or *network-like relationships* (case B3) comprise the four cases (m,m), (m,mc), (mc,m), and (mc,mc).

Instead of the association types, *minimum and maximum thresholds* can be set if deemed more practical. For instance, instead of the multiple association type from projects to employees, a range of (MIN,MAX):=(3,8) could be set. The lower threshold defines that at least three employees must be involved in a project, while the maximum threshold limits the number of participating employees to eight.

[3]The character combination ":=" stands for "is defined by."

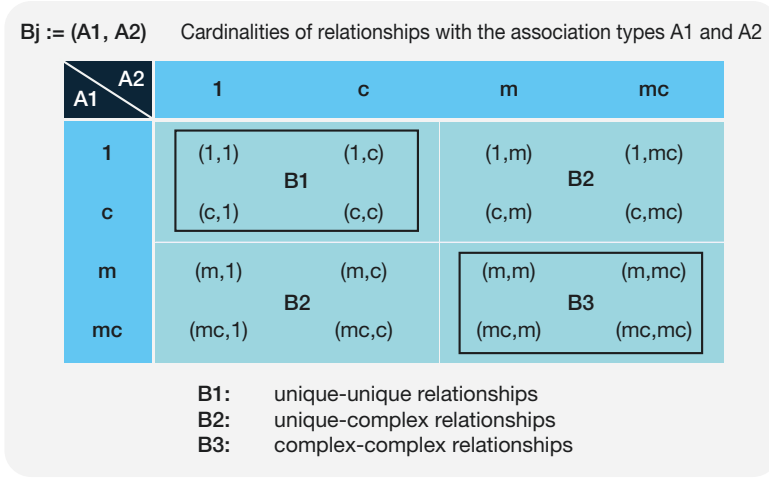


Fig. 2.5 Overview of the possible cardinalities of relationships

2.2.3 Generalization and Aggregation

Generalization is an abstraction process in which entities or entity sets are subsumed under a superordinate entity set. The dependent entity sets or subsets within a generalization hierarchy can, vice versa, be interpreted as *specializations*. The generalization of entity sets can result in various constellations:

- **Overlapping entity subsets**: The specialized entity sets *overlap with each other*. As an example, if the entity set EMPLOYEE has two subsets PHOTO_CLUB and SPORTS_CLUB, the club members are consequently considered employees. However, employees can be active in both the company's photography and sports club, i.e., the entity subsets PHOTO_CLUB and SPORTS_CLUB overlap.
- **Overlapping-complete entity subsets**: The specialization entity sets *overlap with each other and completely cover* the generalized entity set. If we add a CHESS_CLUB entity subset to the PHOTO_CLUB and SPORTS_CLUB and assume that every employee joins at least one of these clubs when starting work at the company, we obtain an overlapping complete constellation. Every employee is a member of at least one of the three clubs, but they can also be in two or all three clubs.
- **Disjoint entity subsets**: The entity sets in the specialization are disjoint, i.e., mutually exclusive. To illustrate this, we will once again use the EMPLOYEE entity set, but this time with the specializations MANAGEMENT_POSITION and SPECIALIST. Since employees cannot at the same time hold a leading position and pursue a specialization, the two entity subsets are disjoint.

- **Disjoint-complete entity subsets**: The specialization entity sets are disjoint, but together completely cover the generalized entity set. As a result, there must be a subentity in the specialization for each entity in the superordinate entity set and vice versa. For example, take the entity set EMPLOYEE with a third specialization TRAINEE in addition to the MANAGEMENT_POSITION and SPECIALIST subsets, where every employee is either part of management, a technical specialist, or a trainee.

Generalization hierarchies are represented by specific forked connection symbols marked "overlapping incomplete," "overlapping complete," "disjoint incomplete," or "disjoint complete."

Figure 2.6 shows the entity set EMPLOYEE as a disjoint and complete generalization of MANAGEMENT_POSITION, SPECIALIST, and TRAINEE. All dependent entities of the entity subsets, such as team leader or department head in MANAGEMENT_POSITION, are also part of EMPLOYEE, since the respective association type is 1. Generalization is, therefore, often called an *is-a relationship*: A team leader *is a(n)* employee, just as a department head *is a(n)* employee. In disjoint complete generalization hierarchies, the reverse association is also of type 1, i.e., every employee is part of exactly one entity subset.

Another important relationship structure beside generalization is *aggregation*, the combination of entities into a superordinate total by capturing their structural characteristics in a relationship set.

To model the holding structure of a corporation, as shown in Fig. 2.7, a relationship set CORPORATION_STRUCTURE is used. It describes the relationship network

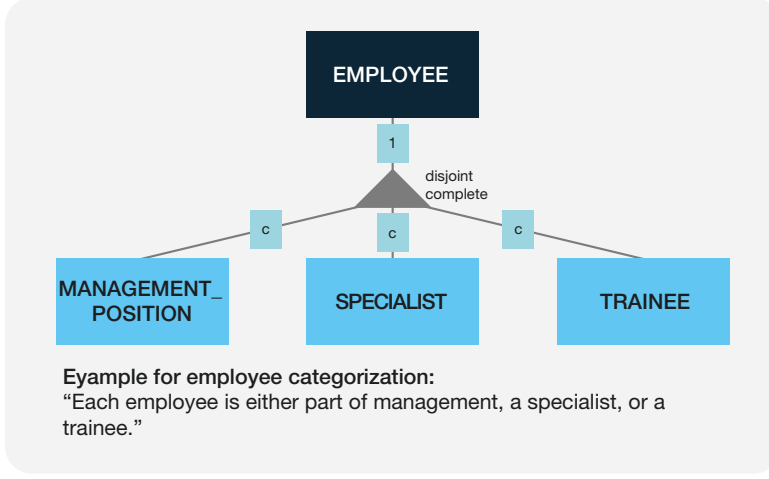


Fig. 2.6 Generalization, illustrated by EMPLOYEE

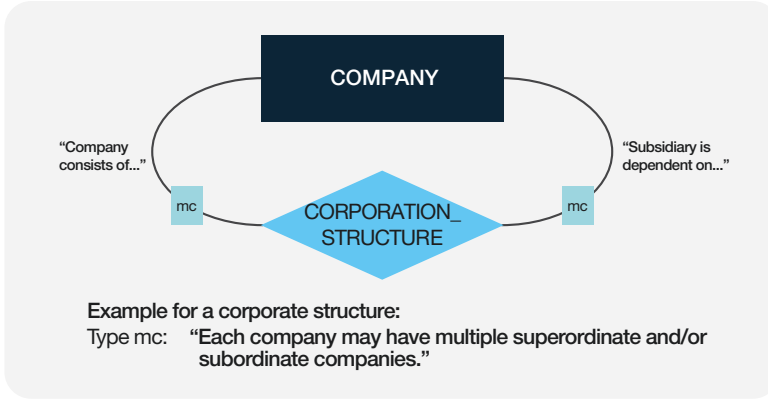


Fig. 2.7 Network-like aggregation, illustrated by CORPORATION_STRUCTURE

of the entity set COMPANY with itself. Each company ID from the COMPANY entity set is used in CORPORATION_STRUCTURE as a foreign key twice, once as ID for superordinate and once for subordinate company holdings (Figs. 2.21 and 2.36). CORPORATION_STRUCTURE can also contain additional relationship attributes such as shares.

In general, aggregation describes the structured merging of entities in what is called a *part-of structure*. In CORPORATION_STRUCTURE, each company can be *part of* a corporate group. Since CORPORATION_STRUCTURE in our example is defined as a network, the association types of both superordinate and subordinate parts must be multiple-conditional.

The two abstraction processes of generalization and aggregation are major structuring elements[4] in data modeling. In the entity-relationship model, they can be represented by specific graphic symbols or as special boxes. For instance, the aggregation in Fig. 2.7 could also be represented by a generalized entity set CORPORATION implicitly encompassing the entity set COMPANY and the relationship set CORPORATION_STRUCTURE.

PART-OF structures do not have to be networks, but can also be hierarchic. Figure 2.8 shows an ITEM_LIST as illustration: Each item can be composed of multiple subitems, while on the other hand, each subitem points to exactly one superordinate item (Figs. 2.22 and 2.37).

The entity-relationship model is very important for computer-based data modeling tools, as it is supported by many CASE (computer-aided software engineering) tools to some extent. Depending on the quality of these tools, both generalization and

[4]Object-oriented and object-relational database management systems support generalization and aggregation as structuring concepts (Chap. 6).

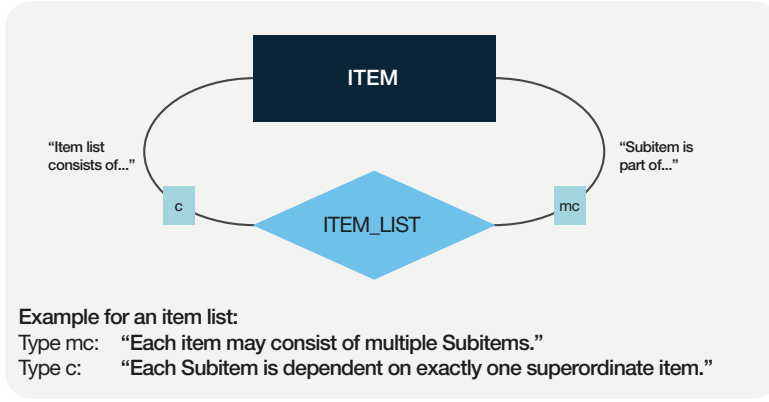


Fig. 2.8 Hierarchical aggregation, illustrated by ITEM_LIST

aggregation can be described in separate design steps, on top of entity and relationship sets. Only then can the *entity-relationship model be converted, in part automatically, into a database schema.* Since this is not always a one-on-one mapping, it is up to the data architect to make the appropriate decisions. Sections 2.3.2 and 2.4.2 provide some simple mapping rules to help convert an entity-relationship model into a relational or graph database.

2.3 Implementation in the Relational Model

2.3.1 Dependencies and Normal Forms

The study of the relational model has spawned a new database theory that describes formal aspects precisely. One of the major fields within this theory is the *normal forms*, which are used to discover and study dependencies within tables in order to avoid redundant information and resulting anomalies.

▶ **About Attribute Redundancy** An attribute in a table is redundant if individual values of this attribute can be omitted *without a loss of information*.

To give an example, the following table DEPARTMENT_EMPLOYEE contains employee number, name, street, and city for each employee, plus their department number and department name.

For every employee of department A6, the table in Fig. 2.9 lists the department name Finances. If we assume that each department consists of multiple employees, similar repetitions would occur for all departments. We can say that the DepartmentName attribute

DEPARTMENT_EMPLOYEE

E#	Name	Street	City	D#	DepartmentName
E19	Stewart	E Main Street	Stow	D6	Accounting
E1	Murphy	Morris Road	Kent	D3	IT
E7	Howard	Lorain Avenue	Cleveland	D5	HR
E4	Bell	S Water Street	Kent	D6	Accounting

Fig. 2.9 Redundant and anomaly-prone table

is redundant, since the same value is listed in the table multiple times. It would be preferable to store the name going with each department number in a separate table for future reference instead of redundantly carrying it along for each employee.

Tables with redundant information can lead to *database anomalies*, which can take one of three forms: If, for organizational reasons, a new department A9, labeled marketing, is to be defined in the DEPARTMENT_EMPLOYEE table from Fig. 2.9, but there are no employees assigned to that department yet, there is no way of adding it. This is an *insertion anomaly*—no new table rows can be inserted without a unique employee number.

Deletion anomalies occur if the removal of some data results in the inadvertent loss of other data. For example, if we were to delete all employees from the DEPARTMENT_EMPLOYEE table, we would also lose the department numbers and names.

The last kind are *update anomalies* (or modification anomalies): If the name of department A3 were to be changed from IT to Data Processing, each of the department's employees would have to be edited individually, meaning that although only one detail is changed, the DEPARTMENT_EMPLOYEE table has to be adjusted in multiple places. This inconvenient situation is what we call an update anomaly.

The following paragraphs discuss normal forms, which help to avoid redundancies and anomalies. Figure 2.10 gives an overview over the various normal forms and their definition. Below, we will take a closer look at different kinds of dependencies and give some practical examples.

As can be seen in Fig. 2.10, the normal forms progressively limit acceptable tables. For instance, a table or entire database schema in the third normal form must meet all

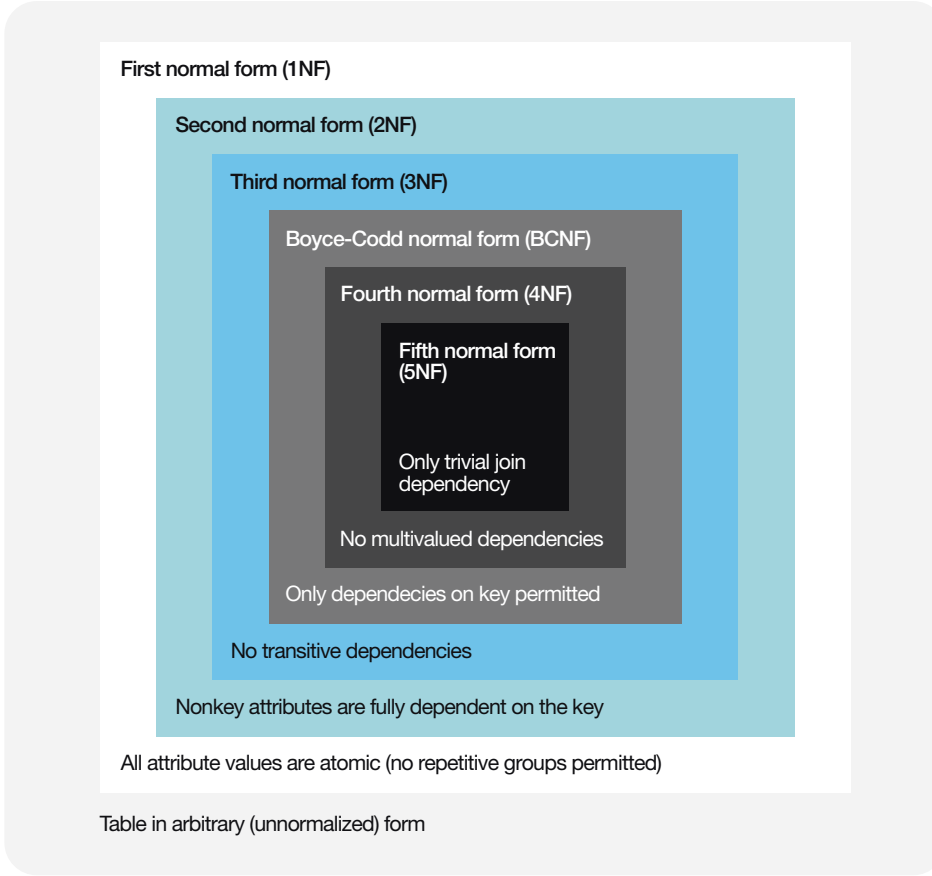


Fig. 2.10 Overview of normal forms and their definitions

requirements of the first and second normal forms, plus there must not be any transitive dependencies between nonkey attributes.

It is important to note that not all normal forms are equally relevant. *Usually only the first three normal forms are used*, since multivalued and join dependencies rarely occur in practice and corresponding use cases are hard to come by. We will, therefore, only take a quick look at the fourth and fifth normal forms.

Understanding the normal forms helps to make sense of the mapping rules from an entity-relationship model to a relational model (Sect. 2.3.2). In fact, we will see that with a properly defined entity-relationship model and consistent application of the relevant mapping rules, the normal forms will always be met. Simply put, by creating an entity-relationship model and using mapping rules to map it onto a relational database schema, *we can mostly forgo checking the normal forms for each individual design step*.

Functional dependencies
The first normal form is the basis for all other normal forms and is defined as follows:

▶ **First normal form (1NF)** A table is in the first normal form when the *domains of the attributes* are atomic. The first normal form requires that each attribute get its values from an unstructured domain, and there must be no sets, lists, or repetitive groups within the individual attributes.

The table PROJECT_PARTICIPANT in Fig. 2.11 is not yet normalized, since each employee tuple contains multiple numbers of projects the employee is involved in. The unnormalized table can be converted to the first normal form by simply creating a separate tuple for each project participation. This conversion of the PROJECT_PARTICIPANT table to 1NF requires the key of the table to be expanded, since we need both the employee and the project number to uniquely identify each tuple. It is common (but not required) with concatenated keys to put the key parts next to each other at the beginning of the table.

Paradoxically, using the first normal form leaves us with a table full of redundancies—in our example in Fig. 2.11, both the names and addresses of the employees are redundantly repeated for each project involvement. This is where the second normal form comes into play:

▶ **Second normal form (2NF)** A table is in the second normal form when, in addition to the requirements of the first normal form, each nonkey attribute is *fully functionally dependent* on each key.

An attribute B is functionally dependent on an attribute A if for each value of A, there is exactly one value of B (written as A->B). A functional dependency of B on A, therefore, requires that each value of A uniquely identifies one value of B. As was seen before, it is a property of identification keys that all nonkey attributes are uniquely dependent on the key, so for an identification key K and an attribute B in one table, there is a functional dependency K->B.

For concatenated keys, this functional dependency must become a full functional dependency: An attribute B is fully functionally dependent on a concatenated key consisting of K1 and K2 (written as (K1,K2)->B) if B is functionally dependent on the entire key, but not its parts, i.e., full functional dependency means that only the entire concatenated key uniquely identifies the nonkey attributes. While the functional dependency (K1,K2)->B must apply, neither K1->B nor K2->B are allowed. *Full functional dependency* of an attribute from a key prohibits a functional dependency of the attribute from any part of the key.

The PROJECT_PARTICIPANT table in 1NF in Fig. 2.11 contains the concatenated key (E#,P#), i.e., it must be tested for full functional dependency. For the names and addresses of the project participants, the functional dependencies (E#,P#)->Name and

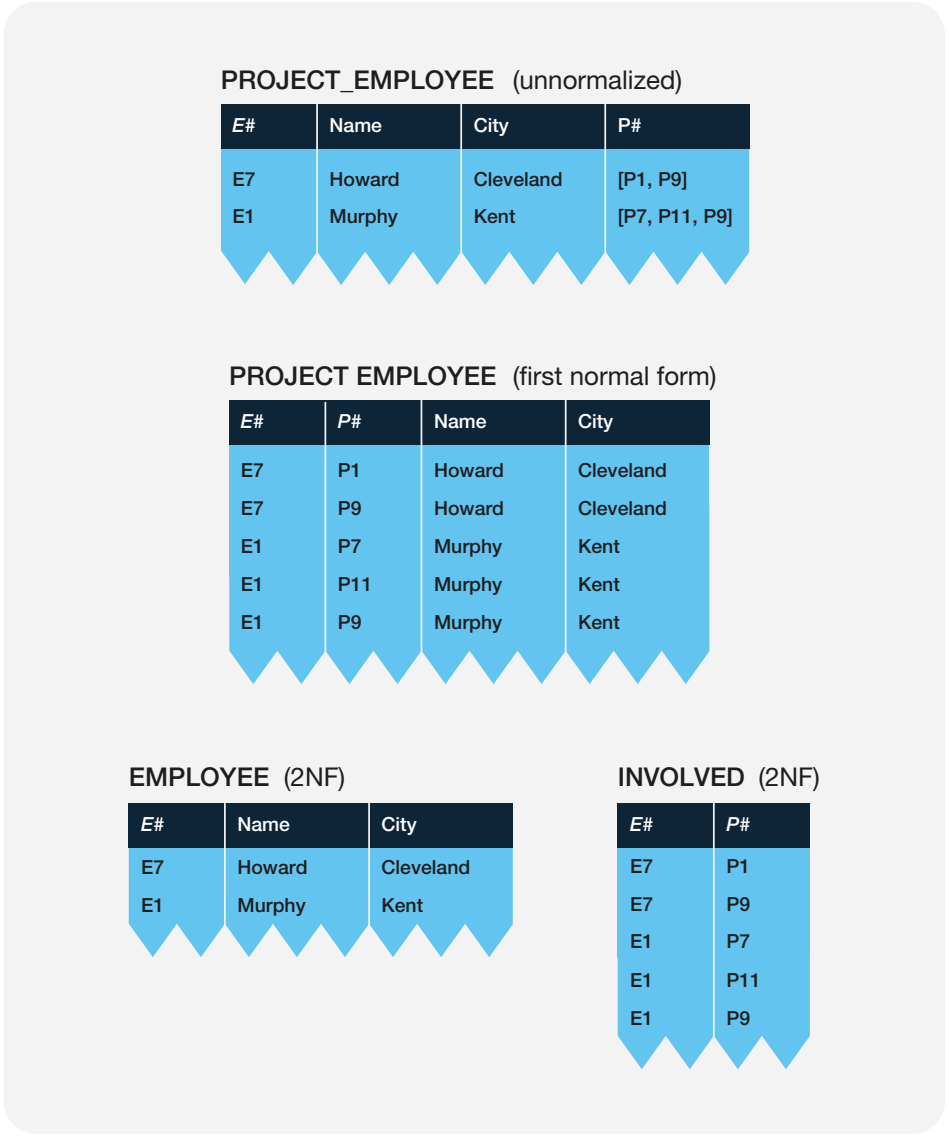


Fig. 2.11 Tables in first and second normal forms

(E#,P#)->City apply. However, while each combination of employee and project number uniquely identifies one name or place of residence, the project numbers have absolutely no bearing on this information, it is already defined by the employee numbers alone. Both the Name and the City attribute are, therefore, functionally dependent on a part of the key, because E#->Name and E#->City. This violates the definition of full functional

dependency, i.e., the PROJECT_PARTICIPANT table is not yet in the second normal form.

If a table with a concatenated key is not in 2NF, it has to be split into subtables. The attributes that are dependent on a part of the key are transferred to a separate table along with that key part, while the concatenated key and potential other relationship attributes remain in the original table.

In our example from Fig. 2.11, this results in the tables EMPLOYEE and PROJECT_INVOLVEMENT, both of which fulfill both the first and the second normal forms. The EMPLOYEE table does not have a concatenated key, and the requirements of the second normal form are obviously met. The PROJECT_INVOLVEMENT table has no nonkey attributes, which saves us the need to check for 2NF here as well.

Transitive dependencies

In Fig. 2.12, we return to the DEPARTMENT_EMPLOYEE table from earlier, which contains department information in addition to the employee details. We can immediately tell that the table is in both first and second normal form—since there is no concatenated key, we do not even have to check for full functional dependency. However, the DepartmentName attribute is still redundant. This can be fixed using the third normal form.

▶ **Third normal form (3NF)** A table is in the third normal form when, in addition to the requirements of the second form, *no nonkey attribute is transitively dependent on any key attribute*.

Again, we use a dependency to define a normal form: In transitive dependencies, an attribute is *indirectly functionally dependent* on another attribute. For instance, the attribute DepartmentName in our table is functionally dependent on the employee number via the department number. We can see functional dependency between the employee number and the department number, as well as between department number and department name. These two functional dependencies E#->D# and D#->DepartmentName can be merged to form a transitive dependency E#->DepartmentName.

In general, given two functional dependencies A->B and B->C with a common attribute B, the merged dependency A->C will also be functional—if A uniquely identifies the values of B, and B uniquely identifies the values of C, C inherits the dependency on A, i.e., the dependency A->C is definitely functional. It is called transitive if apart from the functional dependencies A->B and B->C, A is not also functionally dependent on B. This gives us the following definition for *transitive dependency*: An attribute C is transitively dependent on A if B is functionally dependent on A, C is functionally dependent on B, and A is not functionally dependent on B.

Since the DepartmentName attribute in the example DEPARTMENT_EMPLOYEE table in Fig. 2.12 is transitively dependent on the E# attribute, the table is by definition

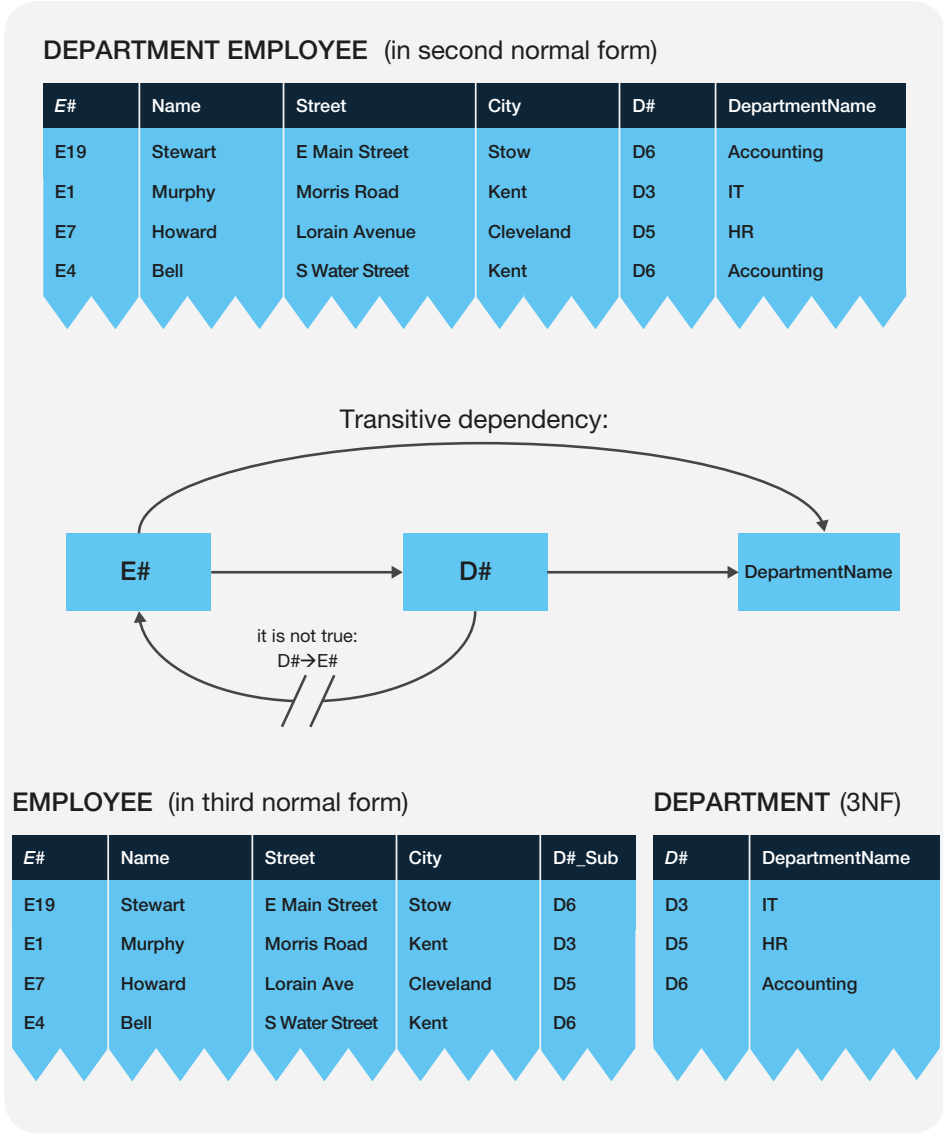


Fig. 2.12 Transitive dependency and the third normal form

not in the third normal form. The transitive dependency can be removed by splitting off the redundant DepartmentName attribute and putting it in a separate DEPARTMENT table with the department numbers. The department number also stays in the remaining EMPLOYEE table as a foreign key with the role SUBORDINATE (see attribute "D#_Sub"). The relationship between employees and departments is, therefore, still ensured.

Multivalued dependencies

The second and third normal forms allow us to eliminate redundancies within the nonkey attributes. However, checking for redundancies must not stop there, since concatenated keys can also exist redundantly.

This should at most require a modification of the third normal form, which is called Boyce-Codd normal form or BCNF after the authors of the major works on it. It is used when there are multiple overlapping candidate keys in one table. Such tables, even when they are in 3NF, may conflict with BCNF. In this case, the table has to be split due to the candidate keys. For more detailed information, see the literature recommendations in Sect. 2.7.

Another normal form results from studying multivalued dependencies between individual key attributes. Although these play only a small role in practice, Fig. 2.13 gives a simple example to quickly illustrate the issue. The original table METHOD is unnormalized, since there may be multiple authors and terms listed next to each method. For instance, the structogram method contains the authors Nassi and Shneiderman while also giving multiple values (sequence, iteration, and branching) for the Term attribute.

We convert the unnormalized table to the first normal form by breaking up the sets {Nassi, Shneiderman} and {sequence, iteration, branching}. The new table consists of only key attributes, i.e., it is not only in 1NF, but also in second and third normal forms. Yet, despite being in 3NF, the table still shows redundant information. For instance, we can see that for each structogram author the three terms sequence, iteration, and branching are listed, while reversely, for each of the terms, both the authors Nassi and Shneiderman are given. We are, therefore, dealing with paired multivalued dependencies, which have to be eliminated.

For a table with the attributes A, B, and C, multivalued dependencies are defined as follows: An attribute C is *multivaluedly dependent* on an attribute A (written as A->->C) if any combination of a specific value of A with an arbitrary value of B results in an identical set of values of C.

For our example from Fig. 2.13, this shows that the Term attribute is multivaluedly dependent on the Method attribute, i.e., Method->->Term. Combining structogram with the author Nassi returns the same set {sequence, iteration, branching} as combining it with the author Shneiderman. There is also a multivalued dependency between the attributes Author and Method in the direction Method->->Author: By combining a specific method such as the structogram with an arbitrary term, e.g., sequence, we obtain the authors Nassi and Shneiderman, the same result as for the combinations of structogram with the terms iteration or branching.

Multivalued dependencies within a table can cause redundancies and anomalies, so they are eliminated with the help of the fourth normal form:

▶ **Fourth normal form (4NF)** The fourth normal form does not permit *multiple true and distinct multivalued dependencies* within one table.

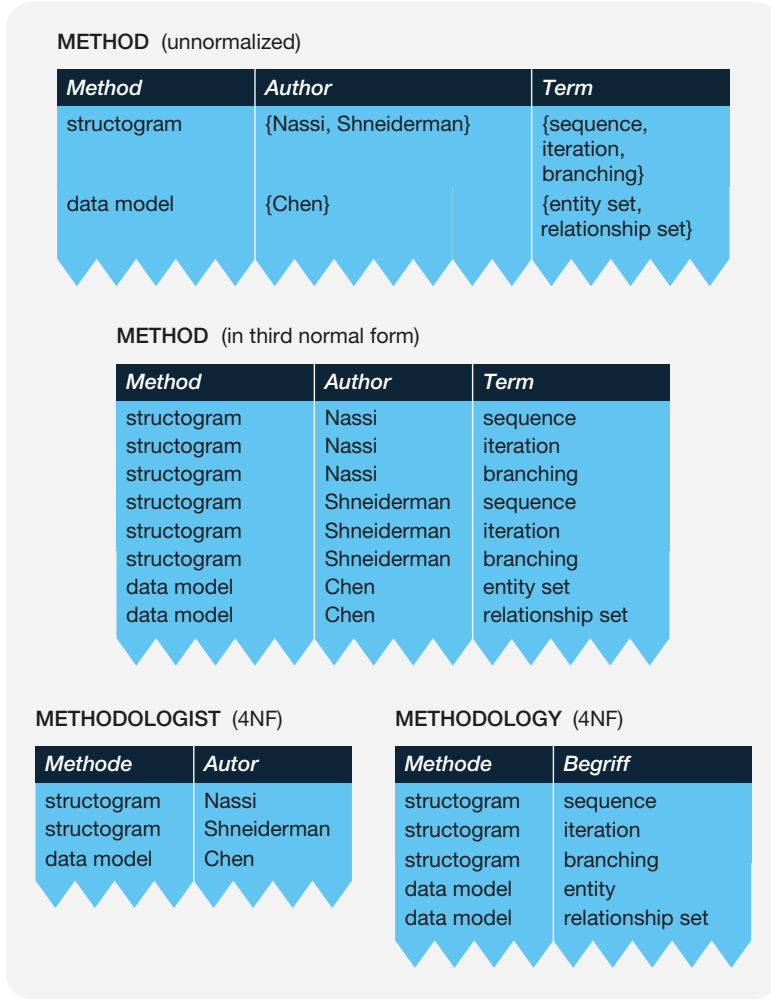


Fig. 2.13 Table with multivalued dependencies

Our METHOD table therefore has to be split into two subtables, METHODOLOGIST and METHODOLOGY. The former relates methods and authors, the latter methods and terms. Both are free of redundancies and adhere to the fourth normal form.

Join dependencies

It is entirely possible for some information to be lost when tables are split into subtables. To avoid this, there are criteria to ensure *lossless table splitting*, i.e., the full preservation of all information via the subtables.

Figure 2.14 shows an improper splitting of the table PURCHASE in the subtables BUYER and WINE. The PURCHASE table contains the wine purchases made by Stewart, Murphy, Howard, and Bell. Combining the subtables derived from it, BUYER and WINE, via the shared attribute Variety returns the INCORRECT PURCHASE table. The information in this table clashes with the original table, since it is indicated that Murphy and Bell have the same taste in wines and both bought the 1996 as well

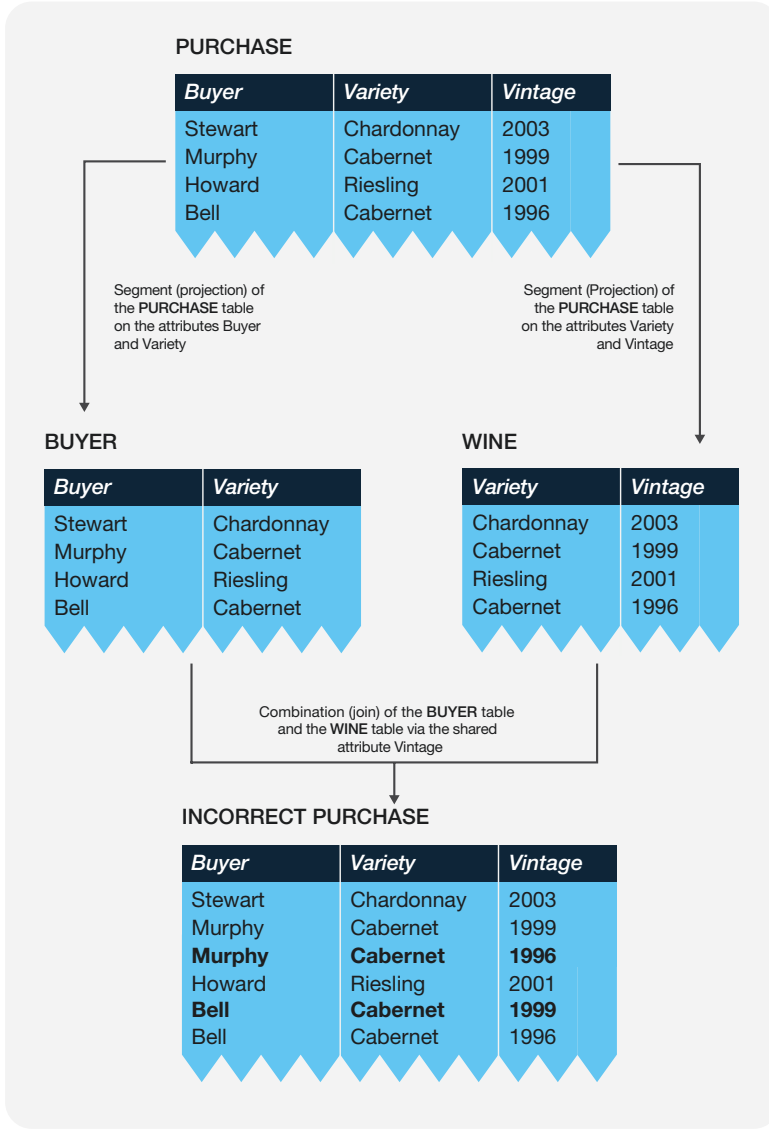


Fig. 2.14 Improper splitting of a PURCHASE table

as the 1999 Cabernet (the erroneous entries for Murphy and Bell are marked in bold in Fig. 2.14).

The BUYER table is split off from the PURCHASE table by reducing the original table to the attributes Buyer and Variety. The required project operator is a filter operator that vertically splits tables into subtables (Fig. 3.3 and Sect. 3.2.3). The WINE table is similarly created by projecting the PURCHASE table onto the two attributes Variety and Vintage. Besides splitting tables into subtables (projection), we can also merge subtables into tables (join operator). Our example in Fig. 2.14 shows how the BUYER and WINE tables can be combined via the shared Variety attribute by complementing all tuples of the BUYER table with the corresponding entries from the WINE table. This gives us the joined table INCORRECT PURCHASE (for more information on joins, see Fig. 3.3 and Sect. 3.2.3), which lists all purchases made by Stewart, Murphy, Howard, and Bell. However, it also contains two purchases that were never actually made—(Murphy, Cabernet, 1996) and (Bell, Cabernet, 1999). Such conflicts can be avoided with the fifth normal form and better knowledge of join dependencies.

▶ **Fifth normal form (5NF)** A table is in the fifth normal form if it can be arbitrarily split by project operators and then reconstructed into the original table with join operators. This property is commonly called *lossless join dependency*. A table is therefore in 5NF when it meets the criterion of lossless join dependency.

The fifth normal form, often called project-join normal form or PJNF, defines how to split a table without issue and, if required, reassemble it without information conflicts. Splitting is done with a project operator and reconstruction with a join operator.

To avoid incorrect information after reconstruction, a table must be checked for join dependencies: A table R with the attributes A, B, and C has join dependency if the projected subtables R1(A,B) and R2(B,C), when joined via the shared attribute B, result in the original table R. This process is termed a lossless join. As illustrated above, the PURCHASE table from Fig. 2.14 is not join dependent and, therefore, not in 5NF.

Figure 2.15 first shows the same table in fourth normal form. Since it does not meet the criterion of join dependency, it has to be converted to the fifth normal form. This is done by creating three subtables from it with the respective projection: BUYER, WINE, and PREFERENCE. A quick test shows that merging those three tables does, indeed, reconstruct the original PURCHASE table. First, the BUYER and WINE tables are combined via the Variety attribute, then another join adds the PREFERENCE table via the Vintage attribute in order to receive the original PURCHASE table.

This discussion certainly invites the question of whether it is possible to use a synthetic approach to database design instead of splitting rules and normal forms (analytical approach). There are, indeed, algorithms that enable the merging of subtables into tables in the third or a higher normal form. Such merging rules allow building database schemas based on a set of dependencies. They formally enable database design to be either top-down (analytical) or bottom-up (synthetic). Unfortunately, very few CASE tools

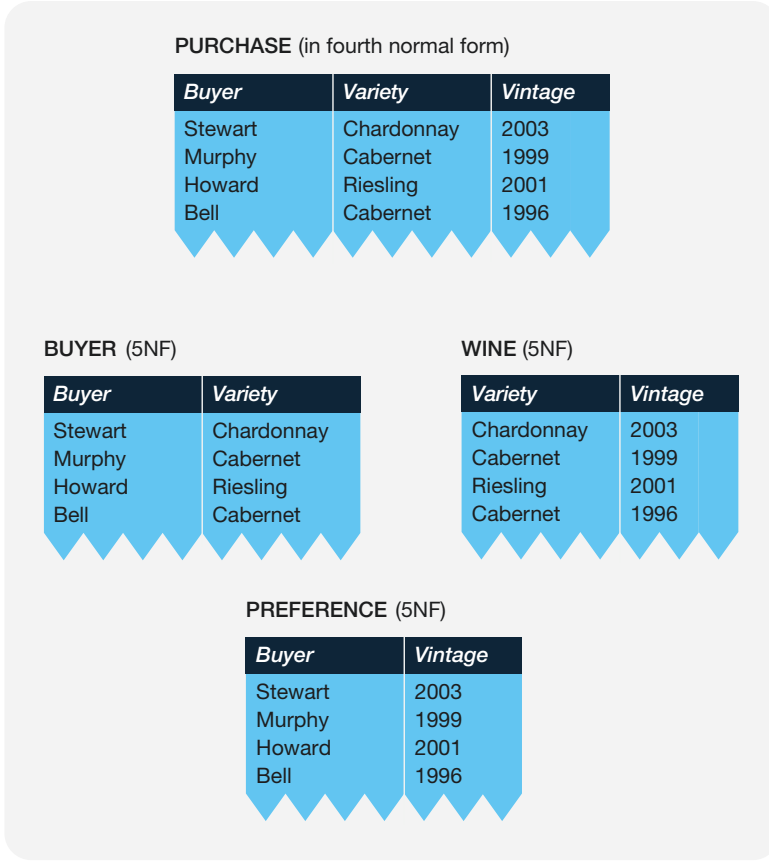


Fig. 2.15 Tables in fifth normal form

support both methods; database architects should, therefore, check the accuracy of their database designs manually.

2.3.2 Mapping Rules for Relational Databases

This section discusses how to map the entity-relationship model onto a relational database schema, i.e., how *entity and relationship sets* can be represented in tables.

A database schema is the description of a database, i.e., the specification of the database structures and the associated integrity constraints. A relational database schema contains definitions of the tables, the attributes, and the primary keys. Integrity constraints set limits for the domains, the dependencies between tables (referential integrity as described in Sect. 2.3.3), and for the actual data.

There are two rules of major importance in mapping an entity-relationship model onto a relational database schema (Fig. 2.16):

▶ **Rule R1 (entity sets)** Each *entity set has to be defined as a separate table* with a unique primary key. The primary key can be either the key of the respective entity set or one selected candidate key. The entity set's remaining attributes are converted into corresponding attributes within the table.

By definition, a table requires a unique primary key (Sect. 1.2.1). It is possible that there are multiple *candidate keys* in a table, all of which meet the requirement of uniqueness and minimality. In such cases, it is up to the data architects which candidate key they would like to use as the primary key.

▶ **Rule R2 (relationship sets)** Each *relationship set can be defined as a separate table*; the identification keys of the corresponding entity sets must be included in this table as

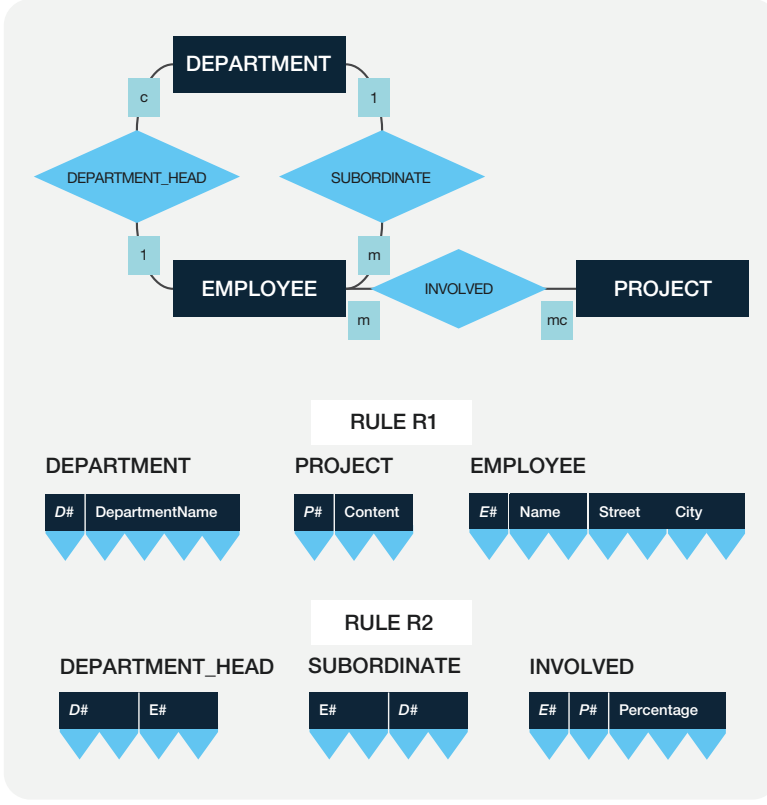


Fig. 2.16 Mapping entity and relationship sets onto tables

foreign keys. The primary key of the relationship set table can be a concatenated key made from the foreign keys or another candidate key, e.g., an artificial key. Other attributes of the relationship set are listed in the table as further attributes.

The term foreign key describes an attribute within a table that is used as an identification key in at least one other table (possibly also within this one). Identification keys can be reused in other tables to create the desired relationships between tables.

Figure 2.16 shows how rules R1 and R2 are applied to a concrete example: Each of the entity sets DEPARTMENT, EMPLOYEE, and PROJECT is mapped onto a corresponding table DEPARTMENT, EMPLOYEE, and PROJECT. Similarly, tables are defined for each of the relationship sets DEPARTMENT_HEAD, SUBORDINATE, and INVOLVED. The DEPARTMENT_HEAD, and SUBORDINATE tables use the department numbers and the employee numbers as foreign keys, while the INVOLVED table takes the identification keys from the EMPLOYEE and PROJECT tables and list the Percentage attribute as another characteristic of the relationship.

Since each department has exactly one department head, the department number D# suffices as identification key for the DEPARTMENT_HEAD table. Likewise, E# can be the identification key of the SUBORDINATE table because each employee belongs to exactly one department.

In contrast, the INVOLVED table requires the foreign keys employee number and project number to be used as a concatenated key, since one employee can work on multiple projects and each project can involve multiple employees.

The use of rules R1 and R2 alone does not necessarily result in an ideal relational database schema as this approach may lead to a high number of individual tables. For instance, it seems doubtful whether it is really necessary to define a separate table for the role of department head in our example from Fig. 2.16. As shown in the next section, the DEPARTMENT_HEAD table is, indeed, not required under mapping rule R5. The department head role would instead be integrated as an additional attribute in the DEPARTMENT table, listing the employee number of the respective head for each department.

Mapping rules for relationship sets

Based on the cardinality of relationships, we can define three mapping rules for representing relationship sets from the entity-relationship model as tables in a corresponding relational database schema. In order to avoid an unnecessarily large number of tables, rule R3 expressly limits which relationship sets *always and in any case require separate tables*:

▶ **Rule R3 (network-like relationship sets)** *Every complex-complex relationship set must be defined as a separate table* which contains at least the identification keys of the associated entity sets as foreign keys. The primary key of a relationship set table is either a concatenated key from the foreign keys or another candidate key. Any further characteristics of the relationship set become attributes in the table.

This rule requires that the relationship set INVOLVED from Fig. 2.17 has to be a separate table with a primary key, which in our case is the concatenated key expressing the foreign key relationships to the tables EMPLOYEE and PROJECT. The Percentage attribute describes the share of the project involvement in the employee's workload.

Under rule R2, we could define a separate table for the SUBORDINATE relationship set with the two foreign keys department number and employee number. This would be useful if we were supporting matrix management and planning to get rid of unique subordination with the association type 1, since this would result in a complex-complex relationship between DEPARTMENT and EMPLOYEE. However, if we are convinced that there will be no matrix management in the foreseeable future, we can apply rule R4 for the unique-complex relationship:

▶ **Rule R4 (hierarchical relationship sets)** *Unique-complex relationship sets can be represented without a separate relationship set table* by the tables of the two associated entity sets. The unique association (i.e., association type 1 or c) allows for the primary key of the referenced table to simply be included in the referencing table as a foreign key with an appropriate *role name*.

Following rule R4, we forgo a separate SUBORDINATE table in Fig. 2.18. Instead of the additional relationship set table, we add the foreign key D#_Subordination to the EMPLOYEE table to list the appropriate department number for each employee. The foreign key relationship is defined by an attribute created from the carried over identification key D# and the role name Subordination.

For unique-complex relationships, including the foreign key can uniquely identify the relationship. In Fig. 2.18, the department number is taken over into the EMPLOYEE

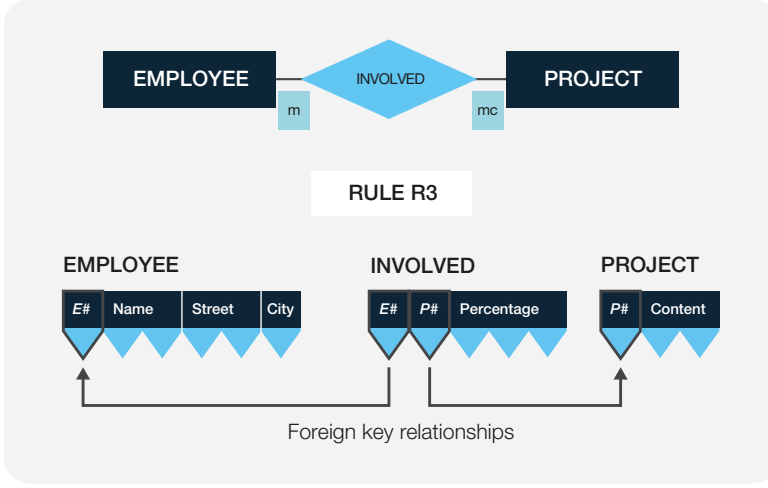


Fig. 2.17 Mapping rule for complex-complex relationship sets

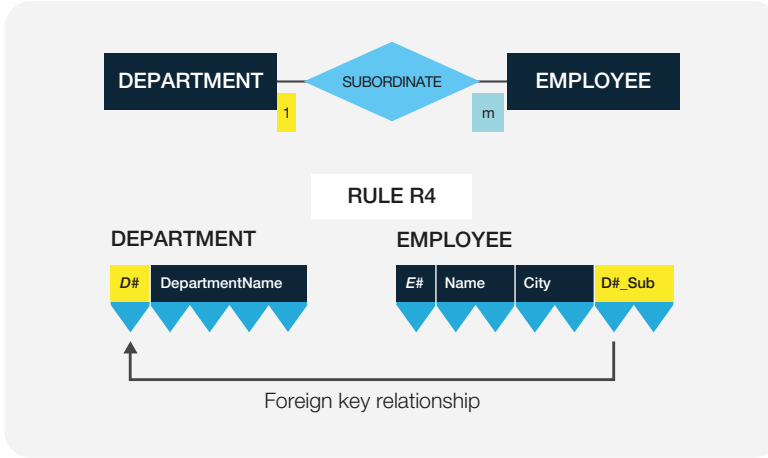


Fig. 2.18 Mapping rule for unique-complex relationship sets

table as a foreign key according to rule R4. If, reversely, the employee numbers were listed in the DEPARTMENT table, we would have to repeat the department name for each employee of a department. Such unnecessary and redundant information is unwanted and goes against the theory of the normal forms (in this case: conflict with the second normal form, see Sect. 2.3.1).

▶ **Rule R5 (unique-unique relationship sets)** *Unique-unique relationship sets can be represented without a separate table* by the tables of the two associated entity sets. Again, an identification key from the referenced table can be included in the referencing table along with a role name.

Here, too, it is relevant which of the tables we take the foreign key from: Type 1 associations are preferable, so the foreign key with its role name can be included in each tuple of the referencing table (avoidance of NULL values, see also Sect. 3.6).

In Fig. 2.19, the employee numbers of the department heads are added to the DEPARTMENT table, i.e., the DEPARTMENT_HEAD relationship set is represented by the M#_DepartmentHead attribute. Each entry in this referencing attribute with the role "DepartmentHead" shows who leads the respective department.

If we included the department numbers in the EMPLOYEE table instead, we would have to list NULL values for most employees and could only enter the respective department number for the few employees actually leading a department. Since NULL values often cause problems in practice, they should be avoided whenever possible, so it is better to have the "DepartmentHead" role in the DEPARTMENT table. For (1,c) and (c,1) relationships, we can, therefore, completely prevent NULL values in the foreign keys, while for (c,c) relationships, we should choose the option resulting in the fewest NULL values.

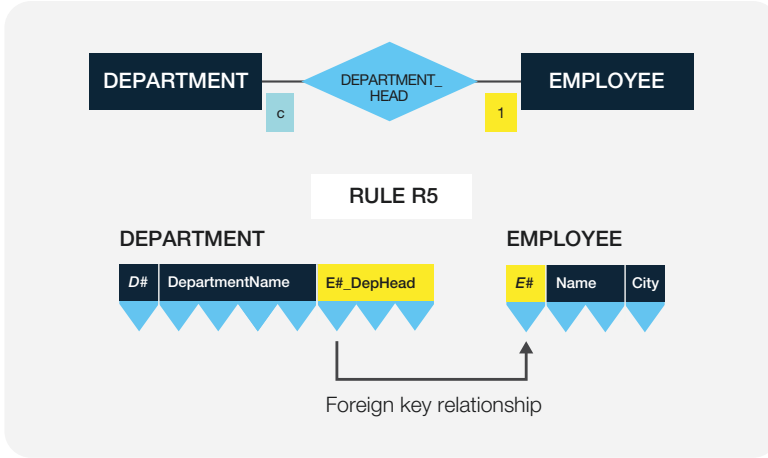


Fig. 2.19 Mapping rule for unique-unique relationship sets

Mapping rules for generalization and aggregation

If there are generalization hierarchies or aggregation structures within an entity-relationship model, they have to be mapped onto the relational database schema as well. Although these special kinds of relationships have the known association types, the corresponding mapping rules are different from those discussed above.

▶ **Rule R6 (generalization)** *Each entity set of a generalization hierarchy requires a separate table*; the primary key of the superordinate table becomes the primary key of all subordinate tables as well.

Since the relational model does not directly support the relationship structure of a generalization, the characteristics of such a relationship hierarchy have to be modeled indirectly. No matter whether a generalization is *overlapping-incomplete, overlapping-complete, disjoint-incomplete, or disjoint-complete*, the identification keys of the specialization must always match those of the superordinate table. Overlapping specializations do not require any special check rules, while disjointness has to be reproduced in the relational model. One way to do this is including an attribute Category in the superordinate table. This attribute mimics class creation and shows which specialization the respective tuple belongs to. On top of that, it must be ensured for any disjoint-complete generalization that there is exactly one entry in a specialization table for each entry in the superordinate table and vice versa.

Figure 2.20 shows a generalization of employee information. Rule R6 results in the tables EMPLOYEE, MANAGEMENT_POSITION, SPECIALIST, and TRAINEE. The tables depending on the EMPLOYEE table must use the same identification key, E#. To avoid individual employees falling into multiple categories simultaneously, we introduce

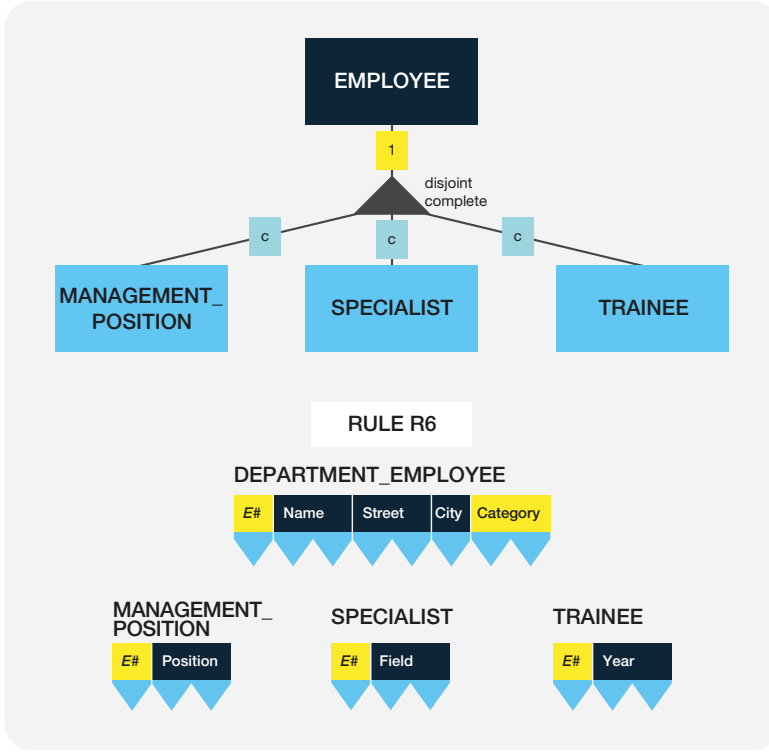


Fig. 2.20 Generalization represented by tables

the Category attribute, which can have the values "Management Position," "Specialist," or "Trainee." This attribute ensures the disjoint characteristic of the generalization hierarchy (Sect. 2.3), which means that the individual entity sets in the specialization must not overlap. The property of completeness cannot be explicitly represented in the database schema and requires a special integrity constraint.

▶ **Rule R7 (aggregation)** If the cardinality of a relationship in an aggregation is *complex-complex*, *separate tables* must be defined for both the entity set and the relationship set. In such cases, the relationship set table contains the identification key of the associated entity set table twice with corresponding role names to form a concatenated key. For *unique-complex* relationships (hierarchical structure), the entity set and the relationship set can be combined in a *single table*.

In the example in Fig. 2.21, the CORPORATION_STRUCTURE has a cardinality of (mc,mc), which means that under rule R7, two tables COMPANY and CORPORATION_STRUCTURE have to be defined. The CORPORATION_STRUCTURE relationship set table shows which companies are direct subsidiaries of a company group and which

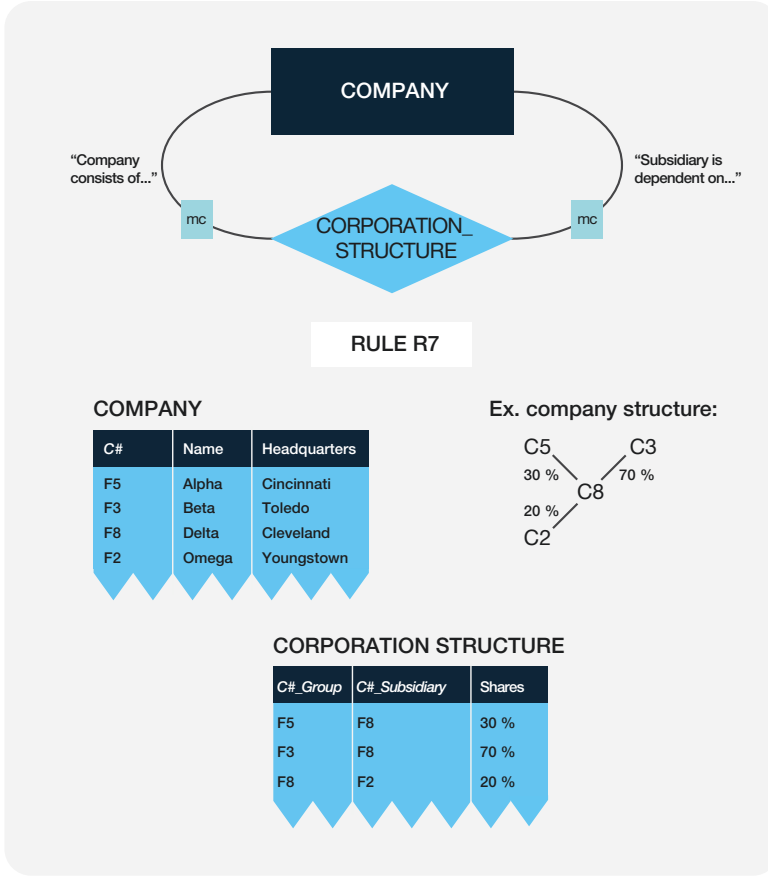


Fig. 2.21 Network-like corporation structure represented by tables

are the directly superordinate group of a certain part of the corporation using the identification keys in each tuple. In addition to the two foreign keys, the CORPORATION_STRUCTURE table contains a Share attribute.

Figure 2.22 illustrates a hierarchical aggregation structure: The ITEM table catalogs the individual components with their material properties, the ITEM_LIST table defines the hierarchical relationship structure within the separate assembly groups. Item A7, for instance, consists of two subitems, A9 and A11, while A11 itself is made from the parts A3 and A4.

If the ITEM_LIST is tree-shaped, i.e., each subitem belongs to exactly one superordinate item, the ITEM and ITEM_LIST tables could be merged into a single ITEM_STRUCTURE table, where the item number of the uniquely superordinate item is listed along with the properties of each item.

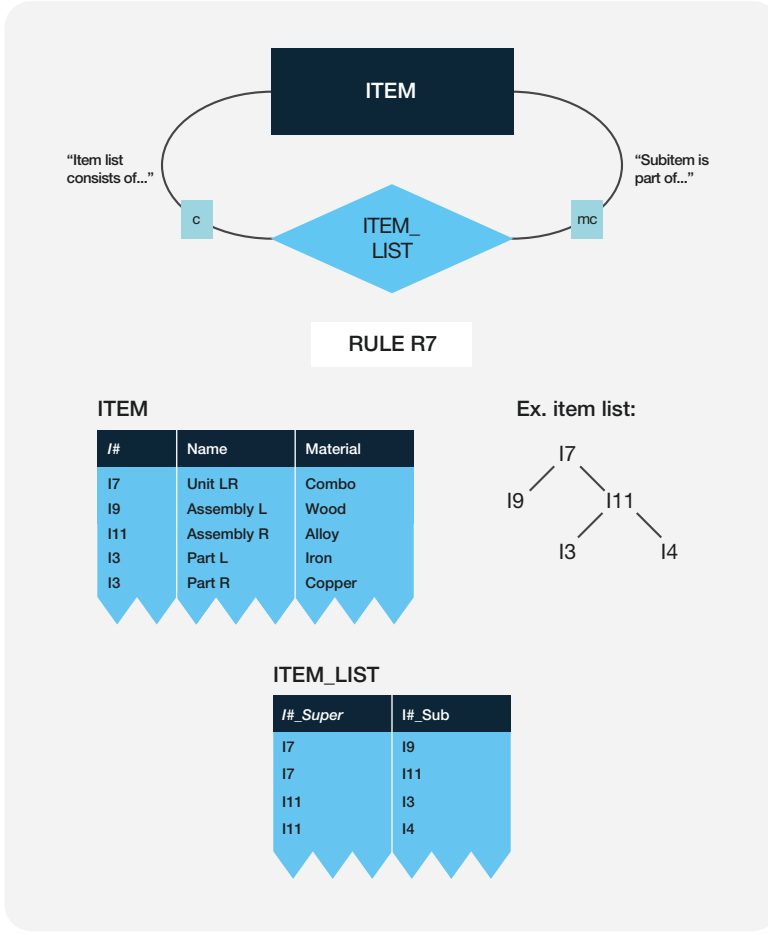


Fig. 2.22 Hierarchical item list represented by tables

2.3.3 Structural Integrity Constraints

Integrity or consistency of data means that stored data does not contradict itself. A database has integrity/consistency if the stored data is free of errors and accurately represents the anticipated informational value. Data integrity is impaired if there are ambiguities or conflicting records. For example, a consistent EMPLOYEE table requires that the names of employees, streets, and cities really exist and are correctly assigned.

Structural integrity constraints are rules to ensure integrity that can be represented within the database schema itself. For relational databases, they include the following:

- **Uniqueness constraint**: Each table has an identification key (attribute or combination of attributes) that uniquely identifies each tuple within the table.
- **Domain constraint**: The attributes in a table can only take on values from a predefined domain.
- **Referential integrity constraint**: Each value of a foreign key must actually exist as a key value in the referenced table.

The *uniqueness constraint* requires a set key for each table. If there are multiple candidate keys within one table, one of them has to be declared the primary key to fulfill the uniqueness constraint. The uniqueness of the primary keys themselves is checked by the DBMS.

The domain constraint, however, cannot be fully achieved by the DBMS—while the domains for individual table columns can be specified, those specifications cover only a small part of the validation rules. Defining a domain is not enough when it comes to verifying city or street names; for instance, a "CHARACTER (20)" limit does not have any bearing on meaningful street or city names. To what extent the contents of a table are validated is at the users' discretion.

A significant help with the domain constraint comes in the form of *enumerated types*, where all possible values of an attribute are entered into a list. Examples of enumerated types are the property categories (attributes) Profession = {Programmer, Analyst, Organizer} or YearOfBirth = {1950…1990}. This type of validation rule is supported by most modern database management systems.

Another major class of validation rules is tied to the term *referential integrity*. A relational database fulfills the referential integrity constraint if each value of a foreign key exists as a value of the referenced primary key. This is illustrated in Fig. 2.23: The DEPARTMENT table has the department number D# as its primary key, which is carried over into the EMPLOYEE table as a foreign key under the D#_Subordination attribute in order to determine which department each employee belongs to. The foreign-primary key relationship has referential integrity if all department numbers listed in the foreign key in the EMPLOYEE table are also present in the primary key of the DEPARTMENT table. In the example in Fig. 2.23, no subordination conflicts with the referential integrity constraint.

However, if we were to try to enter a new tuple "M20, Mahoney, Market Ave S, Canton, D7," the DBMS would reject our insert operation if it supports referential integrity. The value D7 is declared invalid, because it is not listed in the referenced table DEPARTMENT.

Apart from insertion issues, the protection of referential integrity affects other database operations as well. If, for example, a user tries to delete a tuple that is referenced by other tuples in separate tables, the system can react in one of various ways:

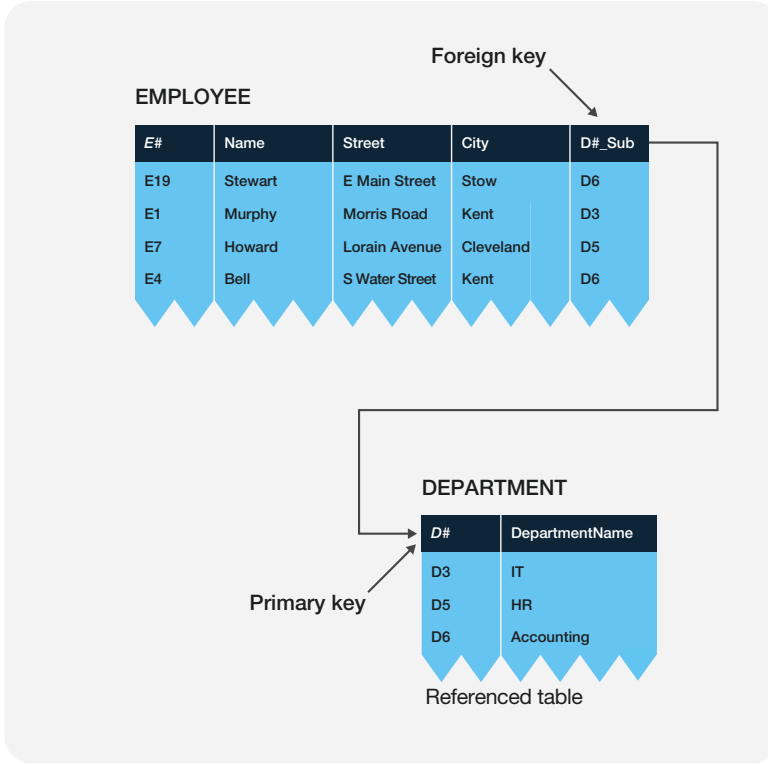


Fig. 2.23 Ensuring referential integrity

▶ **Restricted delete** If a system follows the rule of restricted delete, a deletion operation is not carried out while the tuple to be deleted is referenced by at least one tuple in another table. If we attempted to delete the "D6, Accounting" tuple from the DEPARTMENT table in Fig. 2.23, this operation would be denied under the restrictive delete rule because the employees Stewart and Bell are listed as subordinate to department D6.

As an alternative to restricted delete, the specifications of the EMPLOYEE and DEPARTMENT tables can call for *cascade delete*:

▶ **Cascade delete** This approach means that on deletion of a tuple, *all dependent tuples are removed* as well. In our example in Fig. 2.23, cascade delete would mean that if we deleted the tuple (D6, Accounting), the two tuples (E19, Stewart, E Main Street, Stow, D6) and (E4, Bell, S Water Street, Kent, D6) would be removed from the EMPLOYEE table.

Another deletion rule supporting referential integrity allows for referenced foreign keys to be set to "unknown" during deletion operations. This third deletion rule option is described in more detail in Sect. 3.7 on integrity constraints, after the discussion on handling NULL values in Sect. 3.6. Lastly, manipulation operations can be subject to restrictions that help ensure referential integrity at all times.

2.4 Implementation in the Graph Model

2.4.1 Graph Properties

Graph theory is a complex subject matter vital to many fields of use where it is necessary to analyze or optimize network-like structures. Use cases range from computer networks, transport systems, work robots, power distribution grids, or electronic relays over social networks to economic areas such as CORPORATION_STRUCTUREs, workflows, customer management, logistics, process management, etc.

In graph theory, a graph is defined by the sets of its nodes (or vertices) and edges plus assignments between these sets.

▶ **Undirected graph** An undirected graph G = (V,E) consists of a vertex set V and an edge set E, with each edge being assigned two (potentially identical) vertices.

Graph databases are often founded on the model of directed weighted graphs. However, we are not yet concerned with the type and characteristics of the vertices and edges, but rather the general abstract model of an undirected graph. This level of abstraction is sufficient to examine various properties of network structures, such as:

- How many edges have to be passed over to get from one node to another one?
- Is there a path between two nodes?
- Is it possible to traverse the edges of a graph visiting each vertex once?
- Can the graph be drawn two-dimensionally without any edges crossing each other?

These fundamental questions can be answered using graph theory and have practical applications in a wide variety of fields.

▶ **Connected graph** A graph is *connected* if there are *paths between any two vertices*.

One of the oldest graph problems illustrates how powerful graph theory can be:

The Königsberg bridge decision problem (Eulerian cycles)
In 1736, the mathematician Leonhard Euler discovered, based on the seven bridges in the town of Königsberg (now Kaliningrad), that a path traversing each edge of a graph exactly once can only exist if each vertex has an even degree.

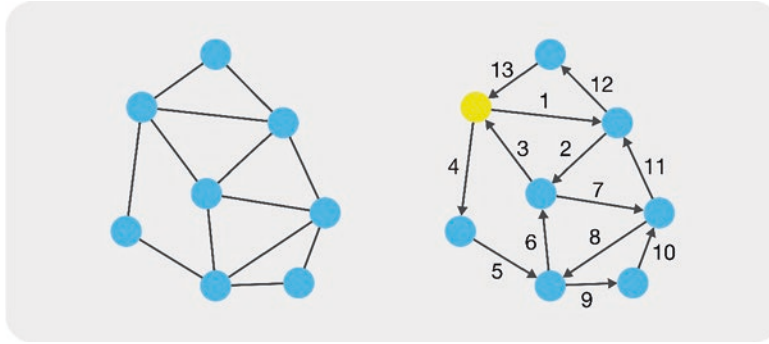


Fig. 2.24 A Eulerian cycle for crossing 13 bridges

▶ **Degree of a vertex** The *degree* of a vertex is the number of edges *incident to it*, i.e., originating from it.

The decision problem for an Eulerian cycle is, therefore, easily answered: A graph G is Eulerian, if it is connected and each node has an even degree.

Figure 2.24 shows a street map with 13 bridges. The nodes represent districts, the edges connecting bridges between them. Every vertex in this example has an even degree, which means that there has to be a Eulerian cycle—but how do we find it?

Fleury's algorithm from 1883 offers simple step-by-step instructions:

▶ **Fleury's algorithm**

1. Choose any node as the starting vertex.
2. Choose any (nonmarked) incidental edge and mark it (e.g., with sequential numbers or letters).
3. Take the end node as the new starting vertex.
4. Repeat from step (2).

In the graph on the left-hand side of Fig. 2.24, this algorithm was used to find a Eulerian cycle. There is, of course, more than one possible solution, and the path does not necessarily have to be a cycle. When does a Eulerian path end at the starting point? The answer can be found by analyzing the degrees of the vertices in Fig. 2.24 in more detail.

This relatively simple example by Euler clearly shows how graph theory can be used to find solutions to various problems. Given any connected graph of arbitrary complexity, we can say that if all its vertices are of an even degree, there is at least one Eulerian cycle.

While Euler's bridge problem looks at edges, the Hamiltonian path problem questions whether all vertices of a graph can be visited exactly once. Despite the similarity of the questions, the Hamiltonian path problem requires more complex algorithms.[5]

[5]It is NP-complete, i.e., it belongs to the class of problems that can be nondeterministically solved in polynomial time.

Dijkstra's algorithm for finding shortest paths
In 1959, Edsger W. Dijkstra published a three-page article describing an algorithm for calculating the shortest paths within a network. This algorithm, commonly called Dijkstra's algorithm, requires a weighted graph (edges weighted, e.g., as distances in meters or minutes) and an initial node from which the shortest path to any other vertex in the network is then determined.

▶ **Weighted graph** *Weighted graphs* are graphs whose vertices or edges have properties assigned to them.

As an example, Fig. 2.25 shows an edge-weighted graph representing a small subway network, with the stops as nodes and the connections between stops as edges. The weights of the edges are the distances between the stops, given in kilometers.

▶ **Weight of a graph** The *weight of a graph* is the *sum of all weights* within the graph, i.e., all node or edge weights.

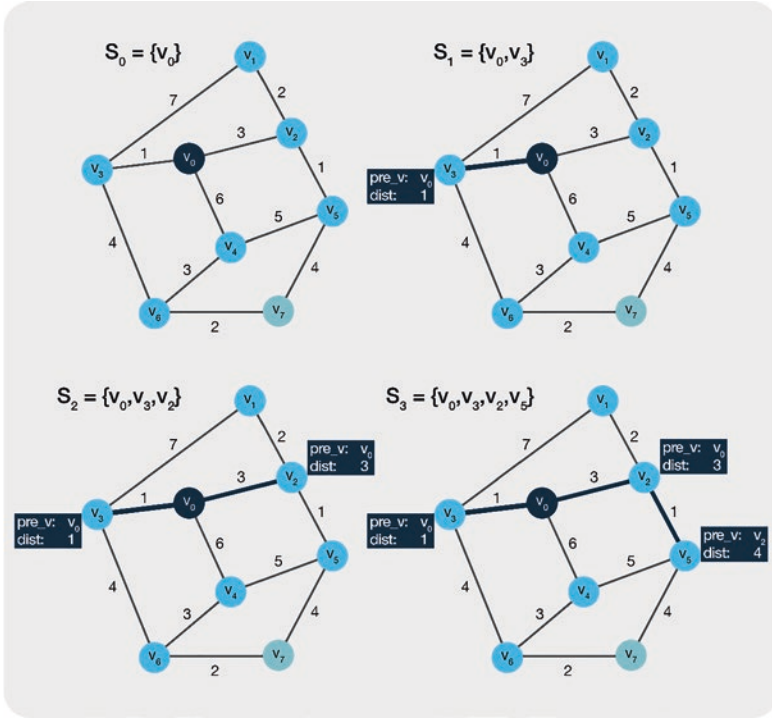


Fig. 2.25 Iterative procedure for creating the set $S_k(v)$

This definition also applies to partial graphs, trees, or paths as subsets of a weighted graph. Of interest is generally the search for partial graphs with maximum or minimum weight. In the subway example of Fig. 2.25, we are looking for the smallest weight between the stations v_0 and v_7, i.e., the shortest path from stop v_0 to stop v_7.

Dijkstra's approach to this problem was to follow the edges indicating the shortest distance from the initial node. Given an undirected graph $G=(V,E)$ with positive edge weights and an initial vertex v_i, we look at the nodes v_j neighboring this vertex and calculate the set $S_k(v)$. We select the neighboring vertex v_j closest to v_i and add it to the set $S_k(v)$.

For the subway example in Fig. 2.25, we start at station v_0 (initial node) and our destination is stop v_7. To initialize, we establish the set $S_k(v)$ as $S_0:=\{v_0\}$. The first step $S_1(v)$ is to inspect all edges connected to v_0, i.e., v_2, v_3, and v_4. By selecting the edge with the shortest distance to v_j (with $j=2$, 3, and 4), we obtain the following set: $S_1=\{v_0,v_3\}$. We proceed in the same manner and examine all edges incident to $S_1(v)$, i.e., v_2, v_4, v_1, and v_6. Since the shortest distance is between v_0 and v_2, we add the vertex v_2 to our set $S_2(v)$ and obtain $S_2=\{v_0,v_3,v_2\}$. Similarly, the next step results in $S_3=\{v_0,v_3,v_2,v_5\}$.

In step $S_4(v)$, we can select either vertex v_1 or vertex v_6, since the paths to both from v_0 are 5 km long. We settle on v_1 and obtain $S_4=\{v_0,v_3,v_2,v_5,v_1\}$. In the same manner, we construct the subsequent sets $S_5=\{v_0,v_3,v_2,v_5,v_1,v_6\}$ and $S_6=\{v_0,v_3,v_2,v_5,v_1,v_6,v_4\}$ and finally $S_7=\{v_0,v_3,v_2,v_5,v_1,v_6,v_4,v_7\}$ with the destination node v_7.

The initial node v_0 is now connected to the destination v_7 by the constructed set $S_7(v)$, and the corresponding route is the shortest possible path. It is 7 km long—1 km from v_0 to v_3, 4 km from v_3 to v_6, and 2 km from v_6 to v_7. Since our solution contains all vertices (or subway stops) in the network, the shortest paths from v_0 to all stations v_i with $i=1,..,7$ can be extrapolated from it.

Figure 2.26 illustrates how Dijkstra's algorithm creates a solution tree (compare the bold connections starting from the initial node v_0 to the tree structure). Each node in the tree is annotated with the previous vertex (pre_v) and the total distance from the start (dist). In v_5, for instance, v_2 is entered as the previous node and 4 (3+1) as the total distance in kilometers from v_0 to v_5.

We can now derive Dijkstra's algorithm for positively weighted graphs, assigning 'Previous vertex' and 'distance' (total distance from the initial node) attributes to each vertex.

The algorithm can be expressed as follows:

▶ **Dijkstra's algorithm**

1. Initialization: Set the distance in the initial node to 0 and in all other nodes to infinite. Define the set $S_{0:}=\{$pre_v: initial node, dist: 0$\}$.
2. Iterate S_k while there are still unvisited vertices and expand the set S_k in each step as described below:
   2a. Calculate the sum of the respective edge weights for each neighboring vertex of the current node.

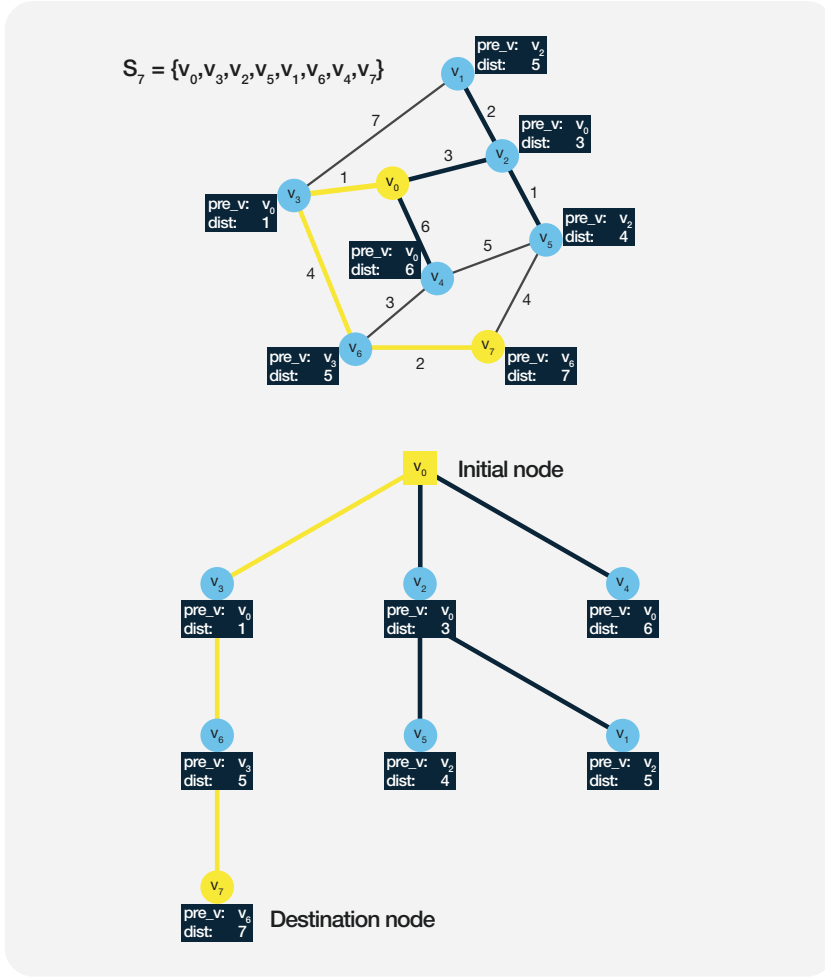


Fig. 2.26 Shortest subway route from stop v_0 to stop v_7

2b. Select the neighboring vertex with the smallest sum.
2c. If the sum of the edge weights for that node is smaller than the distance value stored for it, set the current node as the previous vertex (pre_v) for it and enter the new distance in S_k.

It becomes obvious that with this algorithm, the edges traversed are always those with the shortest distance from the current node. Other edges and nodes are considered only when all shorter paths have already been included. This method ensures that when a

specific vertex is reached, there can be no shorter path to it left (greedy algorithm[6]). The iterative procedure is repeated until either the distance from initial to destination node has been determined or all distances from the initial node to all other vertices have been calculated.

Where is the nearest post office?

A typical use case of graph theory is the search for the nearest post office. In a city with multiple post offices (or clubs, restaurants, movie theaters, etc.), an internet user wants to find the nearest one. Which one is closest to their current location? This is a common example for searches where the answer depends on the user's location, i.e., location-based services.

Given n points (post offices) in the Euclidian plane, i.e., a set $M = \{P_1, P_2, \ldots, P_n\}$, the task is to find the point from M closest to a query point q (location of the user). The quickest and easiest way to do that is to split the city into zones:

▶ **Voronoi diagram** A *Voronoi diagram* divides a plane with a set $M = \{P_1, P_2, \ldots, P_n\}$ of n points into *equivalence classes* by assigning each point P_i its corresponding Voronoi cell V_i.

With Voronoi diagrams, the search for the nearest post office can be simplified to determining the Voronoi cell containing the query location (point location). All points within a Voronoi cell V_i are equivalent in that they are closer to the post office P_i than to any other post offices.

▶ **Voronoi cell** For a set $M = \{P_1, P_2, \ldots, P_n\}$ of points, the Voronoi cell of a point P_i is the set of all points in the Euclidian plane which are *closer to* P_i *than to* P_j for all $j \neq i$.

The construction of a Voronoi cell for a point P_i is rather simple: Take the perpendicular bisectors of the connections to the neighboring points P_j and determine the half-spaces $H(P_i, P_j)$ enclosing the point P_i (Fig. 2.27). The intersection of all half-spaces of P_i forms the Voronoi cell V_i.

Calculating Voronoi diagrams can be complicated, especially if there is a large number of post offices in a large city or urban agglomeration, but there are algorithms that can reduce the computing effort. One method is the Divide et Impera (Latin for divide and rule) approach, in which the problem is repeatedly split into subproblems (Divide) until they are small enough to be easily solved (Impera).

In 1975, Michael Ian Shamos and Dan Hoey proposed an algorithm based on Divide et Impera, which splits the set $M = \{P_1, P_2, \ldots, P_n\}$ in order to recursively obtain the

[6] In each step, greedy algorithms select the locally optimal subsequent conditions according to the relevant metric.

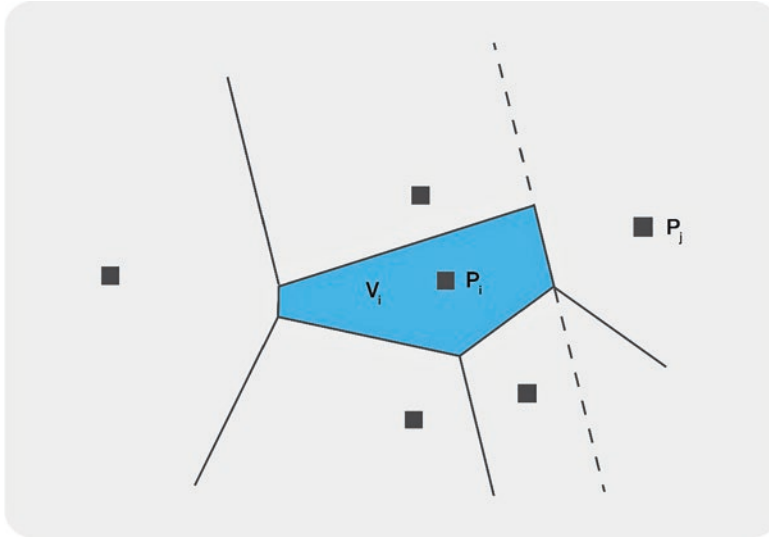


Fig. 2.27 Construction of a Voronoi cell using half-spaces

Voronoi diagram VD(M) from the subdiagrams VD(M_1) and VD(M_2). The major step in this idea is proving that it is possible to merge the two partial solutions VD(M_1) and VD(M_2) in linear time.

Figure 2.28 shows the method for merging two subproblems: Given a dividing line T, let the part of the plane to the right of T be T^+ and the part to the left of T be T^-. The dividing line itself is constructed step-by-step from the convex boundaries of M_1 and M_2. Since the Voronoi cells are convex, the merger between VD(M_1) and VD(M_2) requires a linear effort.

Voronoi diagrams were named after Russian-Ukrainian mathematician Georgy Feodosevich Voronoy, who generalized them for n-dimensional space in 1908. Voronoi diagrams can also be calculated with the help of a graph dual to it: Combining the centers of all Voronoi cells in a diagram yields a Delaunay triangulation as presented by Russian mathematician Boris Nikolaevich Delone in 1934. Because Voronoi diagrams and Delaunay triangulations are dual to each other, all properties of a Voronoi diagram apply to its dual Delaunay graph and vice versa.

Both Voronoi diagrams and Delaunay triangulations are important tools for problem resolution in numerous scientific fields. In IT, they are used for associative searches, cluster computing, planning, simulation, and robotic collision avoidance systems, among others.

Analyzing relationships in social networks

Graph theory can also be used to analyze the relationships between members of a community. A *social network* in this context is a community or group of web users in regular

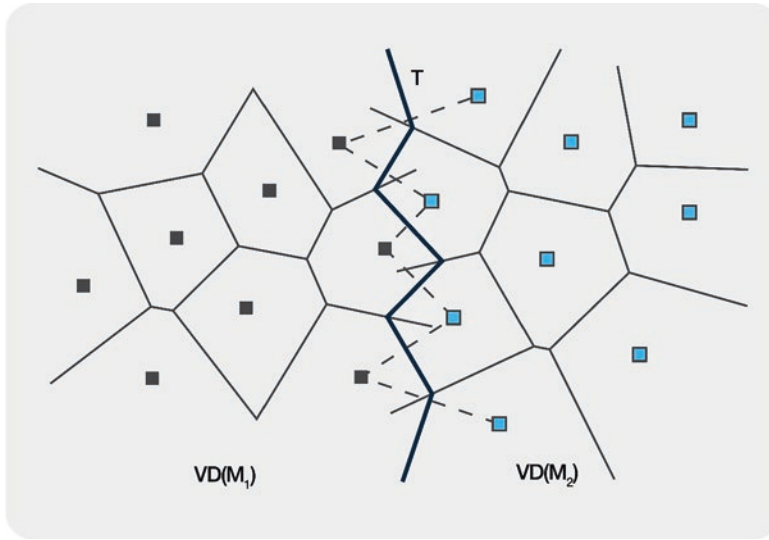


Fig. 2.28 Dividing line T between two Voronoi diagrams $VD(M_1)$ and $VD(M_2)$

social interaction. The group of members determines participation and the needs or interests pursued by the community.

Every social network and each individual member has a *reputation*. Empirical social research uses sociometry to try to understand the relationships between members of a social network. Graph theory, with its various metrics and indices, is a valuable tool for sociologic network analyses.

▶ **Sociogram** A sociogram is a graph representing the relationships within a social group or network. The nodes are individual members, the edges show *how members view* each other.

Indices for the analysis of sociograms refer to the relative position of individual people and the relationships between members. The most important metrics are:

- **Degree**: The degree (or degree centrality) indicates the number of connections from one member to others, i.e., degree centrality equals the degree of the node. The number of outgoing edges determines the vertex' outdegree, the sum of the incoming edges is its indegree. Members with a high degree (called *stars*) stand out, even though degree centrality is not necessarily representative of the person's position or influence.
- **Betweenness**: Betweenness centrality is the number of times a node lies on the shortest path (geodesic distance) between pairs of other members. This measurement takes into account not only the direct, but also the indirect relationships within the social network. For instance, it is possible to calculate through whom most of the information in the network is exchanged.

- **Closeness**: Closeness centrality is an indicator showing the number of edges connecting a person to all other members of the network. It is usually calculated by determining and then averaging the shortest paths to all other members.
- **Density**: The density of a network is the ratio between the number of existing relationships and the maximum possible number of relationships. It is expressed as a decimal value between 0 (no relationships at all) and 1 (all members are interconnected).
- **Clique**: Cliques are groups of at least three members, all of whom have mutual relationships with each other, i.e., each member of a clique interacts with all other members.

Figure 2.29 shows a sociogram of a middle school class, with arrows indicating positive sympathies between students. The lighter nodes represent girls, the darker ones boys. An arrow directed from v_5 to v_{11} indicates that girl #5 finds her fellow student #11 likeable. If girl #11 is friendly with #5 as well, the edge is represented by a double-pointed arrow in the graph.

The sociogram as a directed graph makes it obvious that only a few boys and girls get a lot of sympathies. For example, girl #11 (vertex v_{11}) has five incoming edges, boy #7 (v_7) even six. The most popular students definitely stand out clearly in the sociogram graph.

▶ **Adjacency matrix** An adjacency matrix represents a graph in matrix format and indicates *which vertices are connected, i.e., adjacent, to which edges*. A connected graph of the n-th degree will yield an n x n matrix.

A sociogram can, therefore, be depicted both as a graph and as a matrix, which can then be sorted as needed. At the bottom of Fig. 2.29, we can see the adjacency matrix (who is connected to whom) for the student sociogram. Both representations are equivalent, although the rows and columns of the adjacency matrix were partially sorted. It becomes clear that with students at this age, boys mainly show sympathies for fellow boys (top left quadrant) and girls for fellow girls (bottom right quadrant). Cluster analysis highlights this behavior.

A quick look at the sociogram graph and/or the adjacency matrix shows that only one girl (bottom left quadrant) and one boy (top right quadrant) were brave enough to show sympathies for a student of the other sex. For instance, while girl v_{10} proclaims her sympathy for boy v_7, this is not reciprocated.

The interpretation of sociograms has become a wide and varied field of use. The process is especially helpful in analyzing both individual properties and aspects of group dynamics:

- **Star**: A star in a sociogram is a group member attracting an extraordinary number of positive views (arrows). This characteristic is assigned based on the degree of the node.
- **Isolate**: Members that receive no positive sympathies within a group are called isolates. Boy v_1 and girl v_4, among others, are examples of isolates in Fig. 2.29.

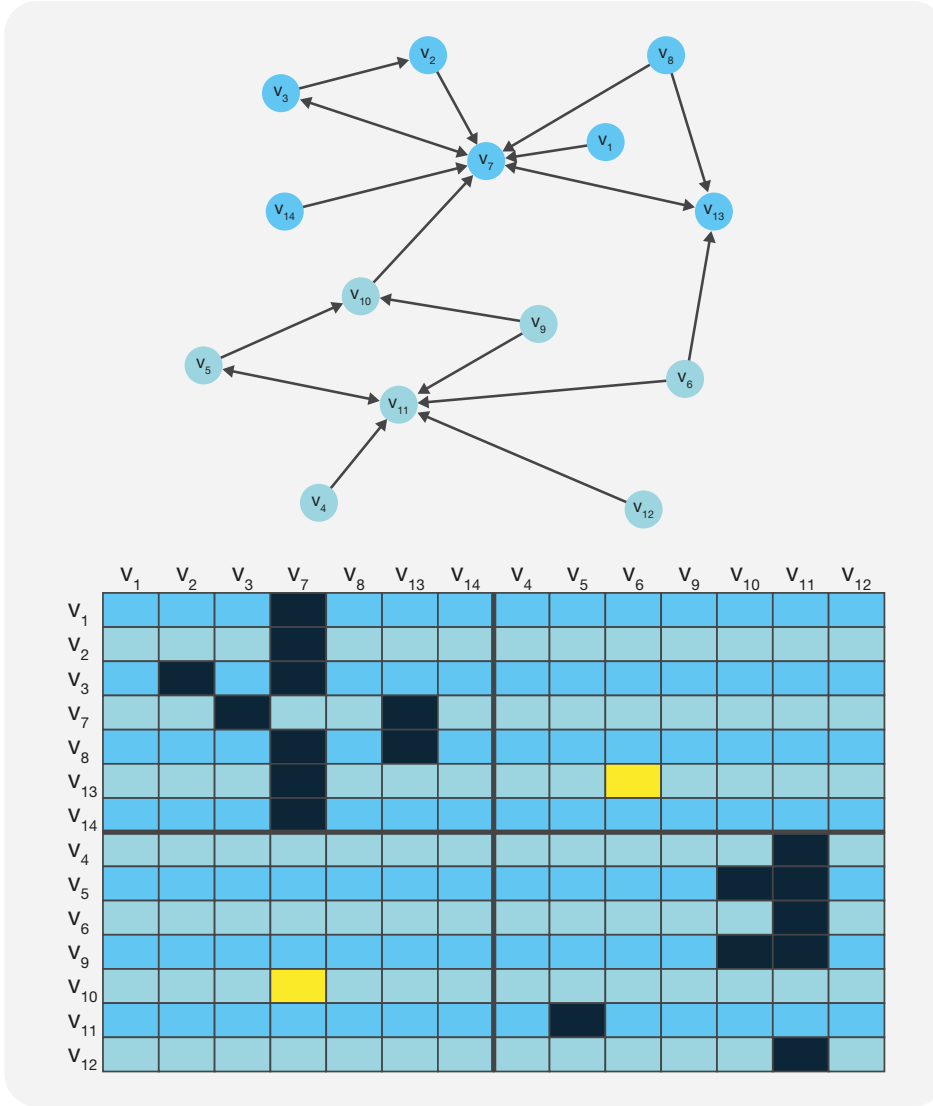


Fig. 2.29 Sociogram of a middle school class as a graph and as an adjacency matrix

- **Ghost**: Ghosts are group members that attract neither positive nor negative views within their group. The term describes how they are barely noticed within the community.
- **Mutual Choice**: Bilateral sympathies are an important factor within communities: The more mutually appreciative relationships exist, the better the social climate within the group. Negative relationships, on the other hand, can impede a community's ability to evolve.

- **Chain**: A chain is formed when one member nominates another member, who nominates another member, and so on. Chains can lead to stars.
- **Island**: Islands are pairs or small subsets of members that are mostly separate from the rest of the community.
- **Triad**: Three group members all assessing each other in mutually positive or negative relationships form a triad. If this structure occurs for more than three members, it is called a circle.

Figure 2.30 shows all possible patterns of triads. Balanced triads (see the left column in Fig. 2.30) are those where all three members hold positive views of each other (case B1) or with one positive and two negative mutual assessments (cases B2, B3, and B4). In case B2, for instance, members A and B hold sympathies for each other, while both have issues with member C, who in turn likes neither A nor B.

The right-hand column of Fig. 2.30 shows the unbalanced triads U1 through U4. Cases U1 through U3 each show two positive pair relationships, which are overshadowed by the third and negative mutual assessment. In U1, A likes both member B and member C (and vice versa), but B and C do not get along. The icy relationship between B and C could put a strain on the connection between A and B and/or A and C.

For communities around famous people, it is common to simply use numbers to characterize the closeness to that person. For instance, we could introduce the fictional Zadeh number (ZN) for Lotfi Zadeh, the founding father of fuzzy logic (see Fuzzy Databases in Sect. 6.8), to indicate how close a fuzzy logic researcher is to the point of origin of the community. Zadeh himself is assigned the Zadeh number 0, researchers who co-authored works with him are assigned 1, researchers who published with them get 2, and so forth. The first generation would consist of Zadeh's students Kosko and Pedrycz (ZN = 1), who co-wrote material with Lotfi Zadeh. The second generation would include, among others, Cudré-Mauroux, Meier, and Portmann (ZZ = 2), who published a research paper with their colleague Pedrycz. Fasel and Kaufmann would then be part of the third generation (ZZ = 3), and so on. Examples of such numbers exist in various communities; actors, for instance, may be assigned a Bacon number indicating whether they worked directly or indirectly on a movie with Kevin Bacon (Bacon number 0).

Graphs or high-performance graph algorithms are commonly used in software applications by employing the methods of linear algebra. If the graph is undirected, there is a symmetrical adjacency matrix. Based on the degrees of the nodes, the respective trees of a graph and various other properties can be deduced. It is possible, for example, to evaluate the relative importance of a node, which is widely used in linking to websites on the internet (page ranking).

In order to determine the reachability of vertices, the n powers of an adjacency matrix are added together (with the identity matrix as the zeroth power). The result is a matrix showing the number of steps needed to reach each node from each other node.

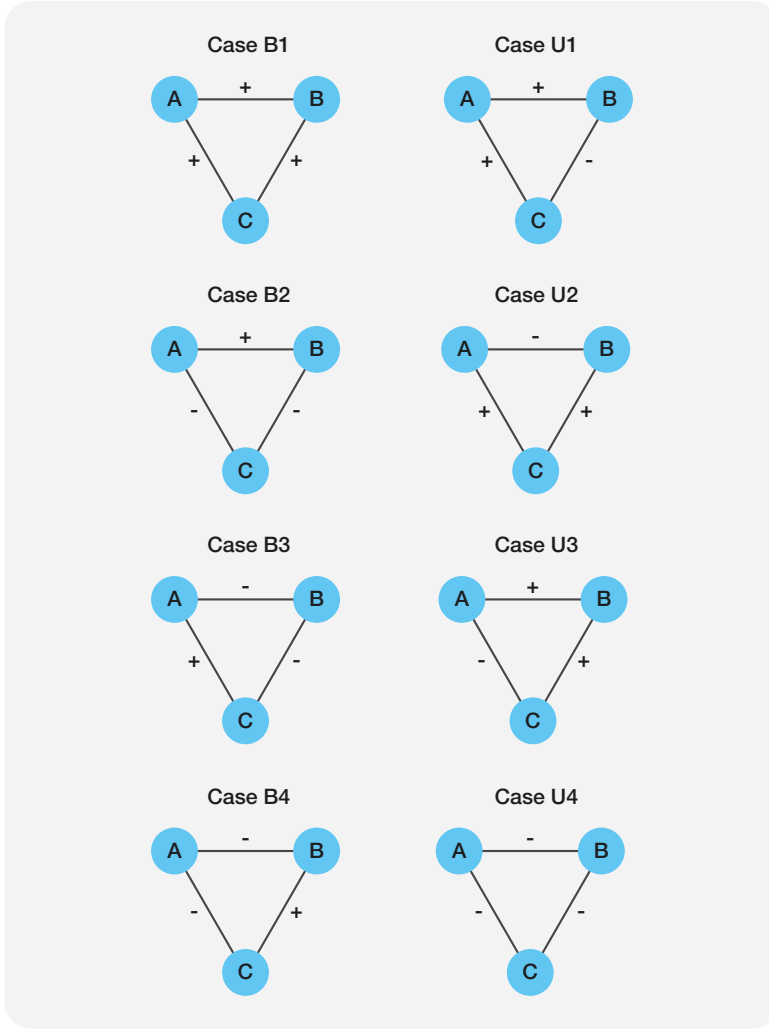


Fig. 2.30 Balanced (B1–B4) and unbalanced (U1–U4) triads

2.4.2 Mapping Rules for Graph Databases

Parallel to the mapping rules R1 to R7 for deriving tables from an entity-relationship model, this section presents the rules G1 to G7 for graph databases. The objective is to convert entity and relationship sets into nodes and edges of a graph.

Figure 2.31 once again shows the previously used project management entity-relationship model (Figs. 2.4 and 2.16). The first mapping rule, G1, concerns the conversion of entity sets into nodes:

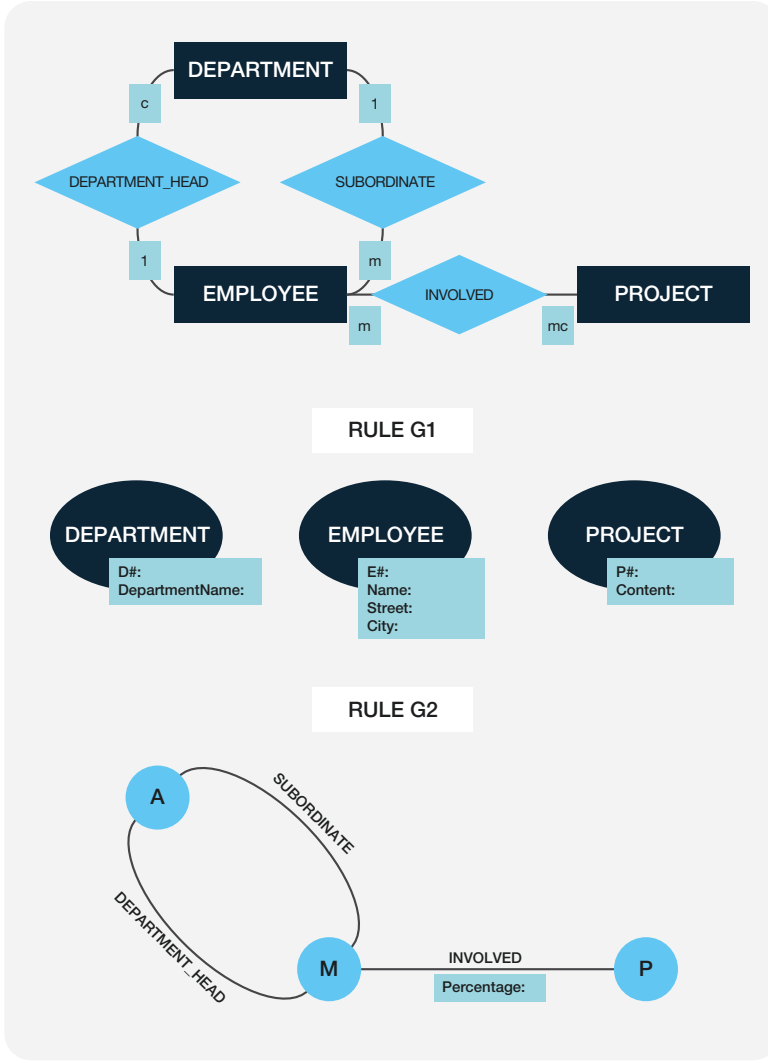


Fig. 2.31 Mapping entity and relationship sets onto graphs

▶ **Rule G1 (entity sets)** *Each entity set has to be defined as an individual vertex* in the graph database. The attributes of each entity set are made into properties of the respective vertex.

The center of Fig. 2.31 shows how the entity sets DEPARTMENT, EMPLOYEE, and PROJECT are mapped onto corresponding nodes of the graph database, with the attributes attached to the nodes (attributed vertices).

Rule G2 governs the mapping of relationship sets onto edges:

▶ **Rule G2 (relationship sets)** *Each relationship set can be defined as an undirected edge* within the graph database. The attributes of each relationship set are assigned to the respective edge (attributed edges).

Applying rule G2 to the relationship sets DEPARTMENT_HEAD, SUBORDINATE, and INVOLVED gives us the following constellation of edges: DEPARTMENT_HEAD and SUBORDINATE between vertices D (for DEPARTMENT), and E (for EMPLOYEE) and INVOLVED between vertices E and P (PROJECT).

Relationship sets can also be represented as directed edges. In the next mapping rules, G3 (for network-like relationships), G4 (hierarchical relationships), and G5 (unique-unique relationships), we will focus on directed edge constellations. They are used to highlight one specific association of a relationship or the direction of the corresponding edge.

Mapping rules for relationship sets

First, we will look at complex-complex or network-like relationships. Figure 2.32 illustrates rule G3, which applies to these constellations.

▶ **Rule G3 (network-like relationship sets)** Any *complex-complex relationship set* can be represented by *two directed edges* where the associations of the relationship provide the names of the edges, and the respective *association types are noted at the arrowheads*. One or both edges can have attributes of the corresponding relationship set attached.

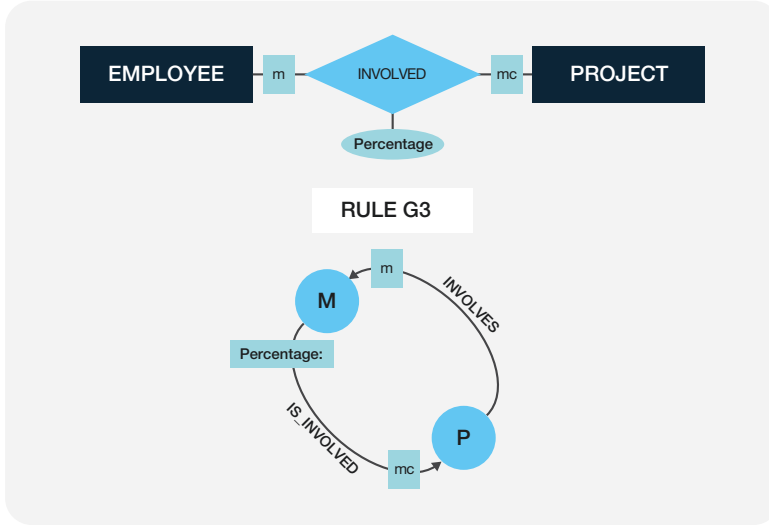


Fig. 2.32 Mapping rule for network-like relationship sets

In Fig. 2.32, rule G3 is applied to the project participation relationship set, resulting in the network-like relationship set INVOLVED being represented by the two edges IS_INVOLVED and INVOLVES. The former goes from the employees (E) to the projects (P) and has the attribute Percentage, i.e., the workload of the individual employees from their assigned projects. Since not necessarily all employees work on projects, the association type 'mc' is noted at the arrowhead. The INVOLVES edge leads from the projects (P) to the employees (E) and has the association type 'm'. As an alternative to the two directed edges, a double arrow could be drawn between the M and P vertices, for instance with the name INVOLVED and the attribute Percentage.

It is also possible to define individual nodes for network-like relationship sets, if desired. Compared to the relational model, the graph model allows for a broader variety of options for representing entity and relationship sets: undirected graph, directed graph, relationship sets as edges, relationship sets as nodes, etc. Rules G3, G4, and G5, however, strongly suggest using directed edges for relationship sets. This serves to keep the definition of the graph database as simple and easy to understand as possible, so that infrequent users can intuitively use descriptive query languages for graphs.

▶ **Rule G4 (hierarchical relationship sets)** *Unique-complex relationship sets* can be defined as *directed edges* between vertices in the *direction from the root node to the leaf node* and with the *multiple association type* (m or mc) noted at the arrowhead.

Figure 2.33 shows the hierarchical subordination of employees of a department. The directed edge HAS_AS_SUBORDINATE goes from the root node D (DEPARTMENT)

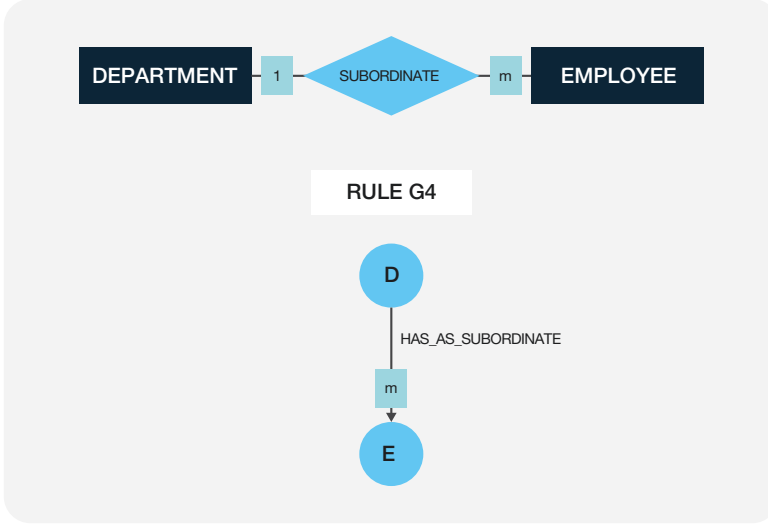


Fig. 2.33 Mapping rule for hierarchical relationship sets

to the leaf node E (EMPLOYEE). The association type at the arrowhead is m, since each department includes multiple employees. Alternatively, the edge could be IS_SUBORDINATE and lead from the employees to the department; however, that would have to have the association type 1, since each employee works in exactly one department.

This leaves us with the case of unique-unique relationship sets:

▶ **Rule G5 (unique-unique relationship sets)** Every *unique-unique relationship set* can be represented as a *directed edge* between the respective vertices. The direction of the edge should be chosen so that the *association type* at the arrowhead is *unique, if possible*.

For instance, Fig. 2.34 illustrates the definition of department heads: The relationship set DEPARTMENT_HEAD becomes the directed edge HAS_AS_DEPARTMENT_HEAD leading from the DEPARTMENT node (D) to the EMPLOYEE node (E). The arrowhead is annotated with '1', since each department has exactly one department head. Of course, it would also be possible to use the reverse direction from employees to departments as an alternative, where the edge would be IS_DEPARTMENT_HEAD and the association type 'c' would be noted at the arrowhead.

The graph-based model is highly flexible and offers lots of options, since it is not limited by normal forms. However, users can use this freedom too lavishly, which may result in overly complex, potentially redundant graph constellations. The mapping rules presented for entity sets (G1), relationship sets (G2, G3, G4, and G5), generalization (G6), and aggregation (G7) are guidelines that may be ignored based on the individual use case.

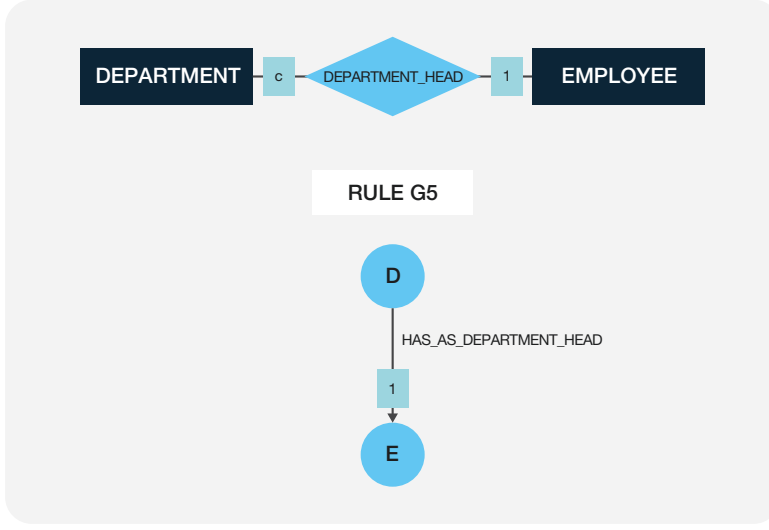


Fig. 2.34 Mapping rule for unique-unique relationship sets

Mapping rules for generalization and aggregation
Below, we will see how to map generalization hierarchies and aggregation networks or hierarchies onto graphs.

▶ **Rule G6 (generalization)** The *superordinate entity set* of a generalization becomes a *double node*, the *entity subsets* become normal *vertices*. The generalization hierarchy is then complemented by *specialization edges*.

Figure 2.35 shows the employees' specializations as management, technical specialists, or trainees. The vertex E is depicted as a double node, representing the superordinate entity set EMPLOYEE. The entity subsets MP (short for MANAGEMENT_POSITION), S (SPECIALIST), and T (TRAINEE) become nodes, with three edges going from E to the MP, S, and T nodes, respectively. The edges are named IS_MANAGEMENT_POSITION, IS_SPECIALIST, and IS_TRAINEE.

The graph-based model can represent generalization hierarchies quite elegantly, since the entity sets involved become nodes, and the IS_A relationships become edges. Unlike in the relational model, it is not necessary to introduce artificial attributes (see the Category

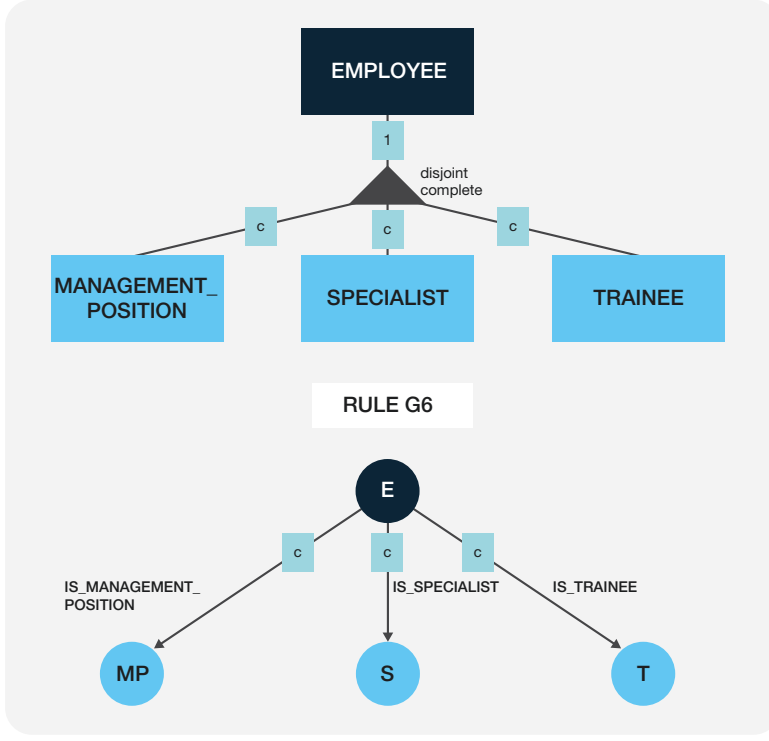


Fig. 2.35 Generalization as a tree-shaped partial graph

attribute in Sect. 2.3.2). We also do not need to define a primary key for the double node and use it as the primary key in the subnodes.

▶ **Rule G7 (aggregation)** For *network-like or hierarchical aggregation structures*, *entity sets are represented by nodes,* and *relationship sets are represented by edges* with the association type mc noted at the arrowhead. Entity set attributes are attached to the nodes; relationship set properties are attached to the edges.

In Fig. 2.36, we see a multicompany CORPORATION_STRUCTURE: The entity set COMPANY becomes the node COMPANY, the relationship set CORPORATION_STRUCTURE becomes the directed edge IS_SUBSIDIARY with mc noted at the arrowhead. The association type mc shows that each company may hold shares of multiple subsidiaries. The attributes of entity and relationship set are converted into properties of the node and the edge, respectively.

Mapping rule G7 can also be used to map hierarchical aggregation structures. For example, Fig. 2.37 contains the ITEM_LIST previously discussed in Sects. 2.2.3 and 2.3.2. The entity set ITEM is represented as the vertex ITEM, the relationship set ITEM_LIST becomes the directed edge CONSISTS_OF. The properties of items and ITEM_LIST are assigned to the vertex and the edge, respectively.

It should have become clear that both generalization hierarchies and aggregation structures (networks, trees) can be stored in a graph database in a simple manner.

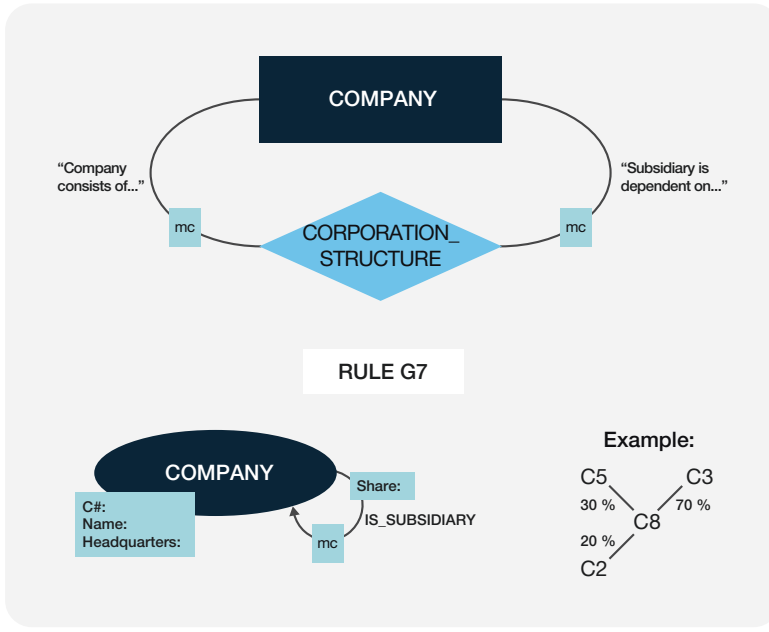


Fig. 2.36 Network-like corporation structure represented as a graph

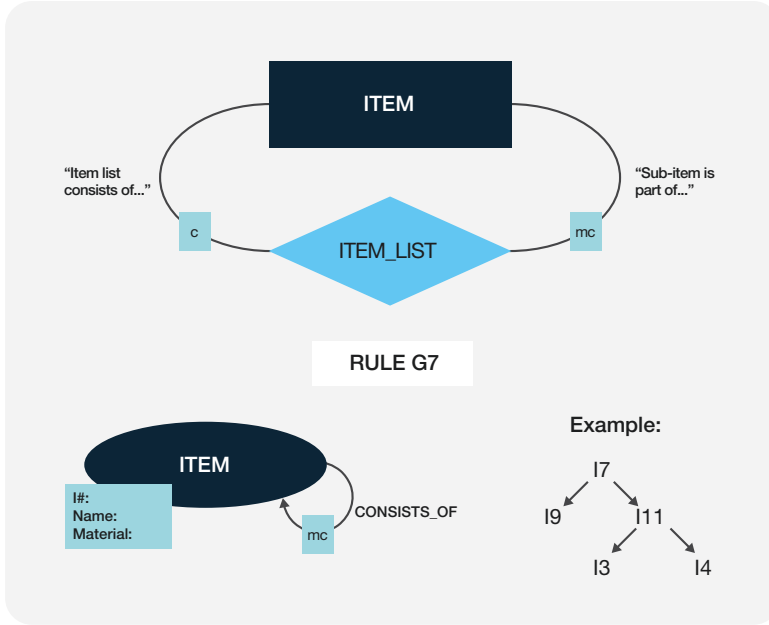


Fig. 2.37 Hierarchical item list as a tree-shaped partial graph

Compared to relational databases, graph databases have the undeniable advantage of allowing the management of various structures such as networks, trees, or chain sequences directly within partial graphs.

2.4.3 Structural Integrity Constraints

Structural integrity constraints exist for both relational databases (Sect. 2.3.3) and graph databases. For graph databases, this means that the graph properties are secured by the database management system itself. Major integrity constraints are:

- **Uniqueness constraint**: Each vertex and each edge can be uniquely identified within the graph. Path expressions (Chap. 3) can be used to navigate to individual edges or nodes.
- **Domain constraint**: The attributes of both vertices and edges belong to the specified data types, i.e., they come from well-defined domains.
- **Connectivity**: A graph is connected if there is a path between any two vertices within the graph. The graph database ensures connectivity for graphs and partial graphs.
- **Tree structures**: Special graphs, such as trees, can be managed by the graph database. It ensures that the tree structure is kept intact in case of changes to nodes or edges.

- **Duality**: Given a planar graph[7] G = (V,E), its dual graph G* = (V*,E*) is constructed by placing a vertex in the dual graph G* for each area of the graph G, then connecting those vertices V* to get the edges E*. Two nodes in G* are connected the by the same number of edges, as the corresponding areas in G have shared edges. A graph database can translate a planar graph G into its dual graph G* and vice versa.

Connectivity, tree structures, and duality concepts are important properties of graph theory. If there is duality, it can be concluded that true statements from a graph translate into true statements of its dual graph and vice versa. This major mathematical tenet is applied in computer science by moving problems to dual space where it is easier to find solution options. If properties can be derived in dual space, they are also valid in the original space.

To give a short example: To solve the post office problem (Sect. 2.4.1), it was necessary to construct a Voronoi cell for each post office by intersecting the half-spaces of the perpendicular bisectors created on the connections between neighboring post offices. In dual space, where straight lines in the plane are converted into points, the intersection of half-spaces can be simplified into a problem of determining the convex hull of a point set[8]. Thanks to existing efficient algorithms for calculating convex hulls, the half-space intersection is thus easily solved.

2.5 Enterprise-Wide Data Architecture

Multiple studies have shown that future users demand complex functions and sophisticated procedures during the definition and creation of information systems but consider the *validity of the data* (Sect. 1.5) much more important when actually using the systems. Data architects are, therefore, advised to answer the following questions first: What data is to be gathered by the company itself and what will be obtained from external data suppliers? How can the stored data be classified and structured according to national and international conditions? Who is in charge of maintaining and servicing the geographically distributed data? What are the relevant obligations regarding data protection and data security in international contexts? Which rights and duties apply to data exchange and disclosure? Those aspects confirm the importance of data architecture for companies and put the appropriate data models in the spotlight.

Due to excessive user demands, analysis and design efforts are nowadays often limited to specific extra features or at best individual fields of use. With SQL and NoSQL

[7]Graphs in the Euclidean plane that have no intersecting edges are called planar graphs.

[8]Kevin Brown utilized dual spaces in his 1979 dissertation on "Geometric Transformations for Fast Geometric Algorithms"; the approach to Voronoi cell construction presented here was proposed by him.

databases, this means a risk of a multitude of stored data only being defined ad hoc or for a local use case. This leads to an *unchecked inflation of SQL and NoSQL databases* with overlapping and ambiguous data, resulting in utter data chaos. Cross-application analyses are impossible or extremely complicated.

Enterprise-wide data architecture helps to avoid this, since it defines the most important entity and relationship sets from a long-term company perspective. It is meant to be superordinate to individual business units and local data views in order to promote a comprehensive view of the company's overarching structures. This data architecture and the data models derived from it form the basis of a well-aligned development of information systems.

Figure 2.38 gives a schematic overview of the connection between cross-department and application-specific data models. An enterprise-wide data architecture describes the data classes required within the company and their relationships with each other. Individual business unit data models are developed on this basis and lead to conceptual database schemas for each application. In practice, such detailing steps cannot be executed strictly top-down, simply because there is not enough time. Especially during changes to existing information systems and the development of new applications, the conceptual database schemas are instead adjusted to the partially available business unit data models and the enterprise-wide data architecture bottom-up, thereby adapting them to the long-term company development step by step.

In addition to the business unit data models that every company has to develop for themselves, there are *industry data model* software products available for purchase. Using such standardization efforts not only reduces the work required to integrate newly

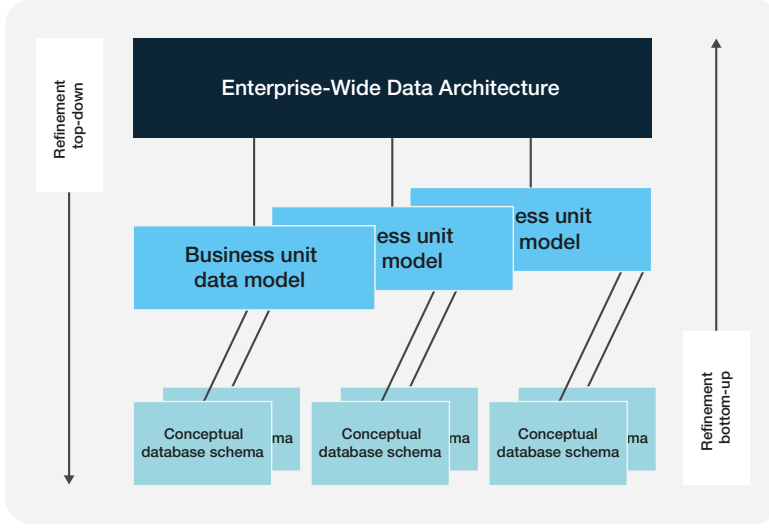


Fig. 2.38 Abstraction steps of enterprise-wide data architecture

purchased application software, but also facilitates the exchange of information across corporate divisions and companies.

As an example, we will look at a limited enterprise-wide data architecture including the entity sets PARTNER, RESOURCE, PRODUCT, CONTRACT, and BUSINESS_CASE.

- **PARTNER**: This includes all natural and legal persons the company has an interest in and about whom information is required to do business. The entity set PARTNER specifically encompasses customers, employees, suppliers, shareholders, government authorities, institutions, and companies.
- **RESOURCE**: This describes raw materials, metals, foreign currencies, securities, or real estate that are offered on the market and purchased, traded, or refined by the company. This entity set can include both material and immaterial goods; a consulting company, for instance, might acquire specific techniques and expertise (Fig. 2.39).
- **PRODUCT**: This set defines the products or services offered by the company. This, too, may consist of material and/or immaterial goods, depending on the industry. Unlike the entity set RESOURCE, PRODUCT characterizes the company-specific development and manufacturing of goods or services.
- **CONTRACT**: This is a legally binding agreement. This entity set comprises insurance, management, and financing agreements, as well as trade, consulting, license, and sales contracts.

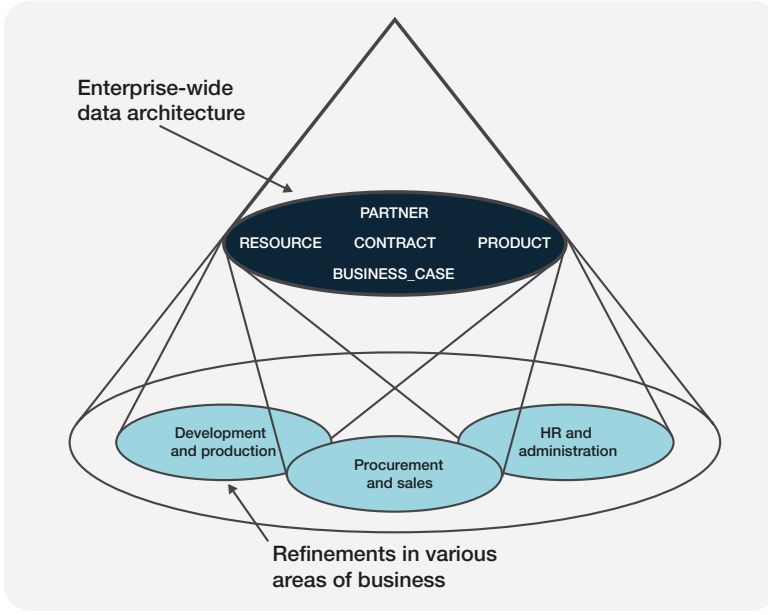


Fig. 2.39 Data-oriented view of business units

- **BUSINESS_CASE**: This is a single step within contract fulfillment that is relevant for business, one event. This may be, for example, a single payment, one booking, one invoice, or one delivery. The entity set BUSINESS_CASE consists of the movements between the other entity sets listed above.

In addition to the generalized entity sets PARTNER, RESOURCE, PRODUCT, CONTRACT, and BUSINESS_CASE, *the most relevant relationships between them from the company's point of view* must be defined. For instance, it can be determined which raw materials are obtained from which partners (supply chain management), who manufactures which company products, or which conditions in a contract apply to which partners and goods.

However, such a bare-bone schematic of an enterprise-wide data architecture is nowhere near enough of a basis to design or develop information systems. Rather, the entity sets and their relationships must be refined step by step, based on individual business units or specific use cases. It is vital in this data-oriented approach that *any realization of an application-specific data model must comply with the enterprise-wide data architecture*. Only then can information systems be developed orderly and in line with the company's long-term goals.

2.6 Formula for Database Design

This section condenses our knowledge of data modeling into a formulaic action plan. Figure 2.40 shows the ten design steps that are taken to a certain extent depending on the project development phase. Practical experience tells us to start by working out the data analysis with a rudimentary entity-relationship model in the *preliminary study*. In the rough or detailed concept, the analysis steps are refined; an SQL, NoSQL, or combined database is created, and the database is tested for consistency and implementation aspects.

The development steps can be characterized as follows: First, as part of data analysis, the *relevant information* must be written into a list. This list can be amended and refined throughout the remaining steps based on input from future users, since the design process is highly iterative. Step 2 is the definition of *entity and relationship sets* and the determination of their identification keys and attribute categories. The resulting entity-relationship model is completed by entering the various association types. In step 3, *generalization hierarchies* and *aggregation structures* in particular can be marked. The fourth step is a comparison and alignment of the entity-relationship model with the *enterprise-wide data architecture* to ensure a coordinated and long-term goal-oriented development of information systems.

In step 5, we map the entity-relationship model onto an *SQL and/or NoSQL database* using the *mapping rules* R1 through R7 and/or G1 through G7 for entity sets, relationship sets, generalization, and aggregation. Step 6 is another refinement of the database

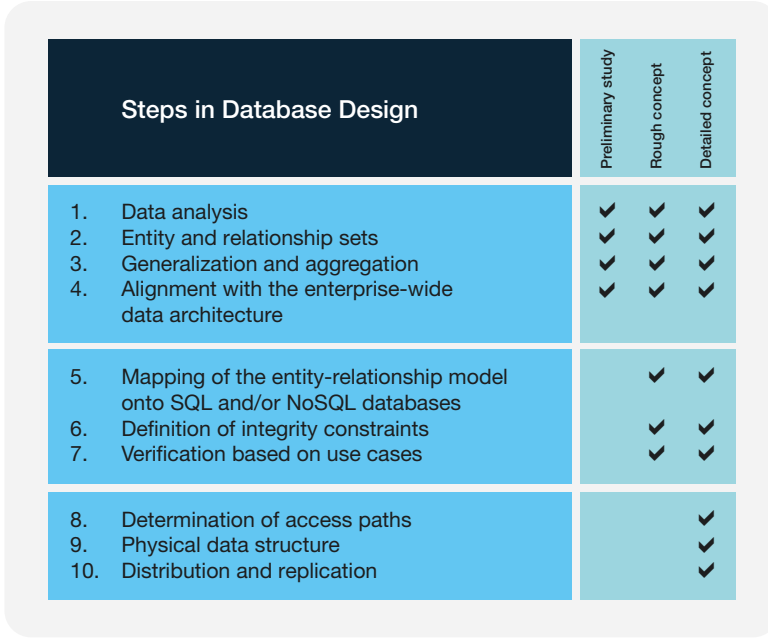


Fig. 2.40 From rough to detailed in ten design steps

design by formulating integrity constraints. These include structural integrity constraints and other validation rules to be implemented with the system in order to ensure that programmers do not have to check consistency individually all the time. In step 7, the database design is checked for completeness based on various specifically developed important use cases (see Unified Modeling Language[9]) prototypically realized with descriptive query languages.

In step 8, the *access paths* for the main application functions are determined. The most common attributes for future database access have to be analyzed and displayed in an access matrix. Step 9 consists of setting the actual quantity structure and defining the *physical data structure*. Finally, the physical distribution of data and the selection of possible replication options make up step 10. When using NoSQL databases, designers have to consider here whether availability and partition tolerance should take priority over strong consistency or not, among other aspects (see also the CAP theorem in Sect. 5.3).

The formula illustrated in Fig. 2.40 focuses largely on data aspects. Apart from the data itself, functions also play a major role in the creation of information systems.

[9]Unified Modeling Language, UML for short, is an ISO-standardized modeling language for the specification, creation, and documentation of software. It allows for easy mapping of an entity-relationship model onto a class diagram and vice versa.

The last years have seen the development of a number of CASE (computer-aided software engineering) tools that support not only database but also function design. Readers interested in the methods of application programming can find additional information in the works listed in the next section.

2.7 Further Reading

The entity-relationship model became widely known due to the works of Senko and Chen (Chen 1976). Since 1979, there have regularly been international conferences on the subject, where additions and refinements to the entity-relationship model are proposed and discussed.

Many CASE tools employ the entity-relationship model for data modeling; however, they lack a common usage of graphic symbols to represent entity sets, relationship sets, or association types, as shown in studies by Balzert (1993), Olle et al. (1988), and Martin (1990). Tsichritzis and Lochovsky (1982) provide an overview of other logical data models.

Blaha and Rumbaugh (2004), Booch (2006), and Coad and Yourdon (1991) discuss object-oriented design. Ferstl and Sinz (1991) are among the German-speaking authors proposing an object-oriented approach to the development of information systems. Balzert (2004) combines methodical aspects for object-oriented analysis by Coad, Booch, and Rumbaugh. A comparison of object-oriented analysis methods can be found in Stein's 1994 work, and Vetter (1998) also promotes an object-oriented approach to data modeling. Hitz et al. (2005) offer an introduction to Unified Modeling Language (UML) especially for software development.

Smith and Smith (1977) established the concepts of generalization and aggregation in the database field. These structural concepts have long been known, especially in the area of knowledge-based systems, e.g., for the description of semantical networks, see Findler (1979).

Research into normal forms gave the database field an actual database theory (Fagin 1979). Major works with a theoretical perspective have been published by Maier (1983), Ullman (1982, 1988), and Paredaens et al. (1989). Dutka and Hanson (1989) provide a concise and easily understandable summary presentation of normal forms. The works of Date (2004), Elmasri and Navathe (2015), Kemper and Eickler (2013), and Silberschatz et al. (2010) dedicate a substantial portion of their content to normalization.

There are several pieces of basic literature on graph theory: Diestel (2006) presents the most important methods and even looks at the Robertson-Seymour theorem[10] that has a fundamental impact on graph theory. Turau (2009) offers an algorithmic approach

[10]The Robertson-Seymour theorem, or graph minor theorem, states that the finite graphs form a well-quasi-ordering due to the graph minor relationship.

to graph theory, while Tittmann (2011) gives an application-oriented introduction to the subject. Works on graph theory in English have been published by Marcus (2008) and van Steen (2010), among others; the latter provides some interesting utilization options for the theory, e.g., computer networks, optimization issues, and social networks. Brüderlin and Meier (2001) show the basics of computer graphics and computational geometry.

Dijkstra (1959) published his algorithm in the journal Numerische Mathematik. Aurenhammer (1991) gives an overview over Voronoi diagrams and basic geometric data structures, whereas Liebling and Pournin (2012) compare Voronoi diagrams and Delaunay triangulations to Siamese twins. Brown (1979) discusses geometric transformations and algorithms in his dissertation and Shamos (1975) wrote his on computational geometry; the recursive algorithm for creating Voronoi diagrams described in Sect. 2.4.1 was published by Shamos and Hoey (1975).

Issues of enterprise-wide data architecture are discussed in Dippold et al. (2005), Meier et al. (1991), and Scheer (1997). The functions and responsibilities of data modeling and data administration are described by Meier and Johner (1991) and Ortner et al. (1990).

References

Aurenhammer, F.: Voronoi Diagrams—A survey of a fundamental geometric data structure. ACM Comput. Surv. **23**(3), 345–405 (1991)

Balzert, H. (Hrsg.): CASE-Systeme und Werkzeuge. Bibliographisches Institut, Mannheim (1993)

Balzert, H.: Lehrbuch der Objektmodellierung—Analyse und Entwurf. Spektrum Akademischer Verlag, Heidelberg (2004)

Blaha, M., Rumbaugh, J.: Object Oriented Modelling and Design with UML. Prentice-Hall, Upper Saddle River (2004)

Booch, G.: Object-Oriented Analysis and Design with Applications. Benjamin/Cummings, Upper Saddle River (2006)

Brown, K.Q.: Geometric Transformations for Fast Geometric Algorithms. PhD Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, USA, 1979

Brüderlin, B., Meier, A.: Computergrafik und Geometrisches Modellieren. Teubner, Wiesbaden (2001)

Chen, P.P.-S.: The entity-relationship model—Towards a unified view of data. ACM Trans. Database Syst. **1**(1), 9–36 (1976)

Coad P., Yourdon E.: Object-Oriented Design. Yourdon Press, Englewood Cliffs (1991)

Diestel R.: Graphentheorie. Springer, Berlin (2006)

Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271 (1959)

Dippold, R., Meier, A., Schnider, W., Schwinn, K.: Unternehmensweites Datenmanagement—Von der Datenbankadministration bis zum Informationsmanagement. Vieweg, Wiesbaden (2005)

Dutka, A. F., Hanson, H. H.: Fundamentals of Data Normalization. Addison-Wesley, Reading (1989)

Elmasri, R., Navathe, S. B.: Fundamentals of Database Systems. Addison-Wesley, Boston (2015)

Fagin, R.: Normal Forms and Relational Database Operators, pp. 153–160. SIGMOD, Proc. Int. Conf. on Management of Data (1979)
Ferstl, O.K., Sinz, E.J.: Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM). Wirtschaftsinformatik **33**(6), 477–491 (1991)
Findler N.V. (ed.): Associative Networks—Representation and Use of Knowledge by Computers. Academic Press, New York (1979)
Hitz M., Kappel G., Kapsammer E., Retschitzegger W.: UML@Work—Objektorientierte Modellierung mit UML2. dpunkt, Heidelberg (2005)
Kemper A., Eickler A.: Datenbanksysteme—Eine Einführung. Oldenbourg, Berlin (2013)
Liebling T. M., Pournin L.: Voronoi Diagrams and Delaunay Triangulations—Ubiquitous Siamese Twins. Documenta Mathematica—Extra Volume ISMP, 2012, pp. 419–431
Maier D.: The Theory of Relational Databases. Computer Science Press, Rockville (1983)
Marcus D. A.: Graph Theory—A Problem Oriented Approach. The Mathematical Association of America, Washington D.C. (2008)
Martin J.: Information Engineering—Planning and Analysis. Prentice-Hall, Englewood Cliffs (1990)
Meier, A., Graf, H., Schwinn, K.: Ein erster Schritt zu einem globalen Datenmodell. Inf. Manag. **6**(2), 42–48 (1991)
Meier, A., Johner, W.: Ziele und Pflichten der Datenadministration. Theor. Prax Wirtschaftsinform. **28**(161), 117–131 (1991)
Olle T. W. et al.: Information Systems Methodologies—A Framework for Understanding. Addison Wesley, Wokingham (1988)
Ortner, E., Rössner, J., Söllner, B.: Entwicklung und Verwaltung standardisierter Datenelemente. Informatik-Spektrum **13**(1), 17–30 (1990)
Paredaens J., De Bra P., Gyssens M., Van Gucht D.: The Structure of the Relational Database Model. Springer, Berlin (1989)
Scheer A.-W.: Architektur integrierter Informationssysteme—Grundlagen der Unternehmensmodellierung. Springer, Berlin (1997)
Shamos M. I., Hoey D.: Closest Point Problem. Proceedings 16th IEEE Annual Symposium Foundation of Computer Science, pp. 151–162 (1975)
Silberschatz A., Korth H. F., Sudarshan S.: Database Systems Concepts. McGraw-Hill, New York (2010)
Smith, J.M., Smith, D.C.P.: Database Abstractions: Aggregation and Generalization. ACM Trans. Database Syst. **2**(2), 105–133 (1977)
Tittmann P.: Graphentheorie—Eine anwendungsorientierte Einführung. Fachbuchverlag Leipzig, München (2011)
Tsichritzis D. C., Lochovsky F. H.: Data Models. Prentice-Hall, Englewood Cliffs (1982)
Turau V.: Algorithmische Graphentheorie. Oldenbourg, Berlin (2009)
Ullman J.: Principles of Database Systems. Computer Science Press, Rockville (1982)
Ullman J.: Principles of Database and Knowledge-Base Systems. Computer Science Press, Rockville (1988)
Vetter M.: Aufbau betrieblicher Informationssysteme mittels pseudo-objektorientierter, konzeptioneller Datenmodellierung. Teubner, Wiesbaden (1998)

Database Languages

3

3.1 Interacting with Databases

Successful database operation requires a database language meeting the different user requirements. Relational query and manipulation languages have the advantage that one and the same language can be used to create databases, assign user permissions, and change or analyze table contents. Graph-based languages provide predefined routines for solving graph problems, e.g., finding the shortest path.

The *database administrator* uses a database language to manage the data descriptions for the company, such as tables and attributes. It is usually helpful if they have a data dictionary system (see glossary definition) to support them. In cooperation with the *data architect*, the system administrator ensures that the description data is consistently managed and kept up to date and in adherence with the enterprise-wide data architecture. Using a suitable CASE[1] tool may prove beneficial here. In addition to governing data formats, the database administrator also sets permissions to restrict both access to data to individual tables or even specific attributes and certain operations, such as deleting or modifying tables, to a limited group of users.

Database specialists define, install, and monitor databases, using system tables specifically geared towards this purpose. These tables make up the system catalog containing all necessary database descriptions and statistics throughout the database management system's runtime. Database specialists can use predefined queries on the system information to get a picture of the current state of all databases without having to deal with the actual tables with user data in detail. For data protection reasons, they should only have access to data during operations under special circumstances, e.g., for troubleshooting (Fig. 3.1).

[1]CASE = computer-aided software engineering.

A. Meier and M. Kaufmann, *SQL & NoSQL Databases,*
https://doi.org/10.1007/978-3-658-24549-8_3

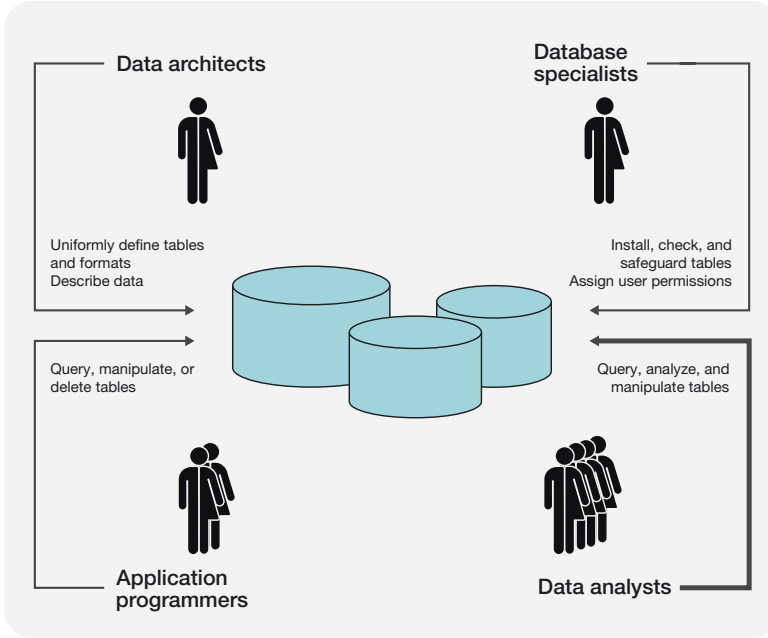


Fig. 3.1 SQL as an example for database language use

Application programmers use database languages to run analyses on or apply changes to databases. Since relational query and data manipulation languages are set-based, programmers require a *cursor concept* (Sect. 3.5.1) to integrate database access into their software development. This enables them to work through a set of tuples record by record in a program. The database language can be embedded into the developed software via APIs. Programmers can also use relational languages to test their applications, including checking their test databases and providing prototypes to future users.

The final user group of database languages are infrequent users and, most importantly, *data analysts,* or data scientists, who need them for their *everyday analysis requirements.* Data analysts are experts on the targeted interpretation of database contents with the help of a database language. Users from the various technical departments who have limited IT skills, but need information regarding specific issues, request a database analysis with an appropriate query phrasing from the data analyst.

This shows how various user groups can use relational database languages to fulfill their work requirements, with data analysts using the same language as application programmers, database administrators, and data architects. Database applications and analyses, as well as technical tasks for securing and reorganizing databases can all be realized in just one language. This means that less training is needed and facilitates the exchange of experiences between user groups.

3.2 Relational Algebra

3.2.1 Overview of Operators

The majority of today's commercially used database management systems are still based on SQL or its derivatives, i.e., languages working with tables and operations on tables. *Relational algebra* provides the *formal framework for the relational database languages*. It defines a number of algebraic operators that always apply to relations. Although most modern relational database languages do not use these operators directly, they are only considered relationally complete languages in terms of the relational model if the original potential of relational algebra is retained.

Below, we will give an overview of the operators used in relational algebra, divided into set operators and relational operators, on two sample relations R and S. Operators work on either one or two tables and always output a new relation. This consistency (algebraic property) allows for the combination of multiple operators and their effects on relations.

Set operators match the known set operations (Fig. 3.2 and Sect. 3.2.2). This group consists of set union with the symbol ∪, set intersection ∩, set difference \, and the Cartesian product ×. Two relations R and S that are union-compatible can be combined (R∪S), intersected (R∩S), or subtracted (R\S). The Cartesian product of two relations R and S (R×S) can be defined without conditions. These set operations result in a new set of tuples, i.e., a new relation.

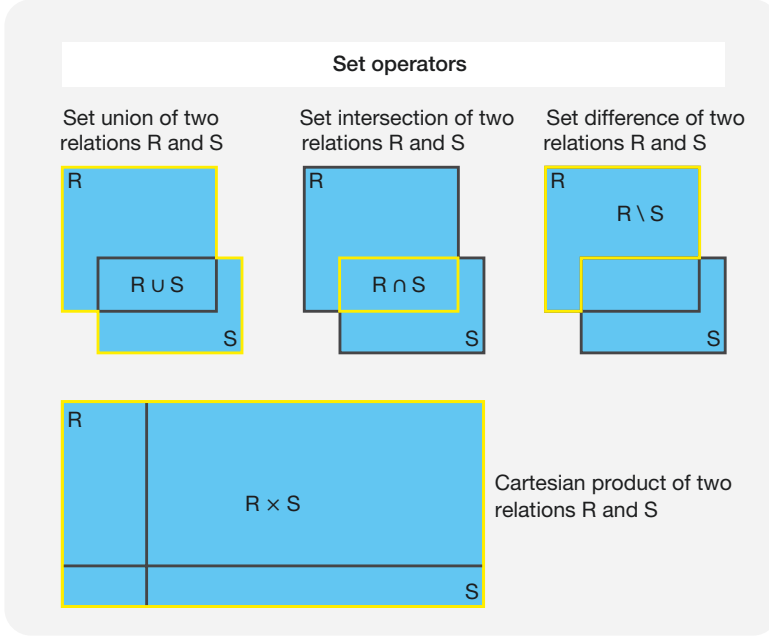


Fig. 3.2 Set union, set intersection, set difference, and Cartesian product of relations

The *relational operators* shown in Fig. 3.3 were defined by Ted Codd specifically for relations and are discussed in detail in Sect. 3.2.3. The project operator, represented by Greek letter π (pi), can be used to reduce relations to subsets. For instance, the expression $\pi_A(R)$ forms a subset of the relation R based on a set A of attributes. An expression $\sigma_F(R)$ with the select operator σ (Greek letter sigma) takes a range of tuples from the relation R based on a selection criterion, or formula, F. The join operator, symbol |×|, conjoins two relations into a new one. For instance, the two relations R and S can be combined by an operation R|×|PS with P specifying the applicable join condition, or join predicate. Lastly, a divide operation R÷S, with the divide operator represented by the symbol ÷, calculates a new table by dividing the relation R by the relation S.

The following two sections provide a more detailed explanation of the set and relational operators of relational algebra with illustrative examples.

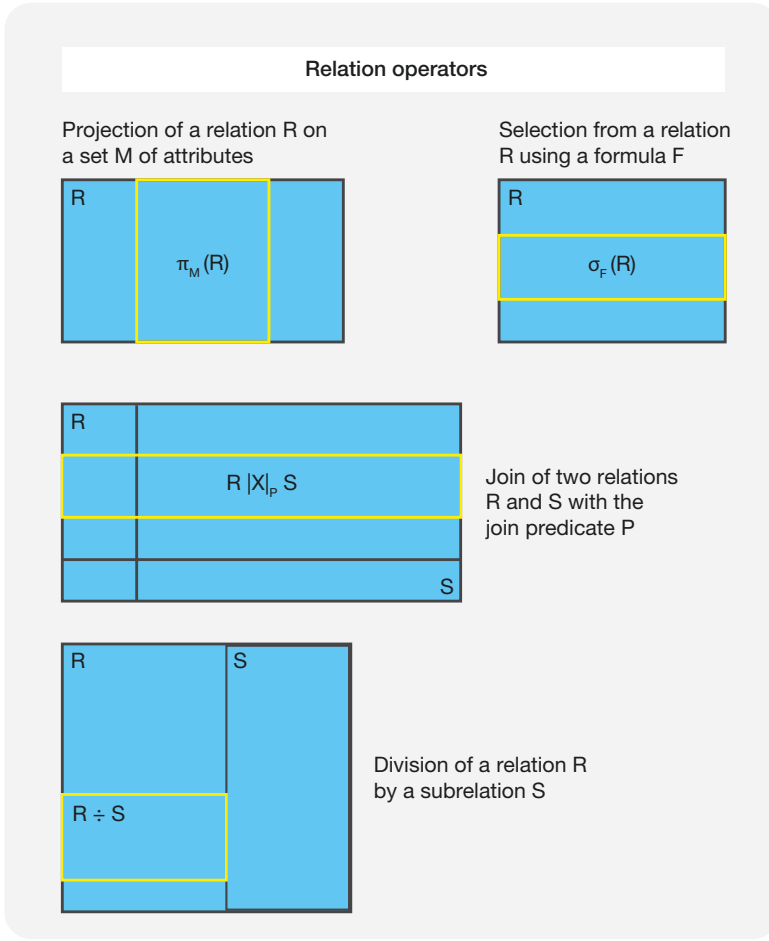


Fig. 3.3 Projection, selection, join, and division of relations

3.2.2 Set Operators

Since every relation is a set of records (tuples), multiple relations can be correlated using set theory. However, it is only possible to form a set union, set intersection, or set difference of two relations if they are union-compatible.

▶ **Union compatibility** Two relations are *union-compatible* if they meet both of the following criteria: Both relations have the same number of attributes and the data formats of the corresponding attribute categories are identical.

Figure 3.4 shows an example: For each of two company clubs a table has been defined from an employee file, containing employee numbers, last names, and street names. The two tables SPORTS_CLUB and PHOTO_CLUB are union-compatible: They have the same number of attributes, with values from the same employee file and therefore defined from the same range.

In general, two union-compatible relations R and S are combined by a *set union* R∪S, where *all entries from R and all entries from S are entered into the resulting table.* Identical records are automatically unified, since a distinction between tuples with identical attribute values in the resulting set R∪S is not possible.

The CLUB_MEMBERS table (Fig. 3.5) is a set union of the tables SPORTS_CLUB and PHOTO_CLUB. Each result tuple exists in the SPORTS_CLUB table, the PHOTO_CLUB table, or both. Club member Howard is only listed once in the result table, since duplicate results are not permitted in the unified set.

SPORTS_CLUB

E#	Name	Street	City
E1	Murphy	Murray Road	Kent
E7	Howard	Lorain Avenue	Cleveland
E19	Stewart	E Main Street	Stow

PHOTO_CLUB

E#	Name	Street	City
E4	Bell	S Water Street	Kent
E7	Howard	Lorain Avenue	Cleveland

Fig. 3.4 Union-compatible tables SPORTS_CLUB and PHOTO_CLUB

CLUB_MEMBERS = SPORTS_CLUB ∪ PHOTO_CLUB

E#	Name	Street	City
E1	Murphy	Murray Road	Kent
E7	Howard	Lorain Avenue	Cleveland
E19	Stewart	E Main Street	Stow
E4	Bell	S Water Street	Kent

Fig. 3.5 Set union of the two tables SPORTS_CLUB and PHOTO_CLUB

The other set operators are defined similarly: The *set intersection* R∩S of two union-compatible relations R and S holds only those entries found in both R and S. In our table excerpt, only employee Howard is an active member of both the SPORTS_CLUB and the PHOTO club.

The resulting set SPORTS_CLUB∩PHOTO_CLUB is a singleton, since exactly one person has both memberships.

Union-compatible relations can also be subtracted from each other: The *set difference* R\S is calculated by removing all entries from R that also exist in S. In our example, a subtraction SPORTS_CLUB\PHOTO_CLUB would result in a relation containing only the members Murphy and Stewart. Howard would be eliminated, since they are also a member of the PHOTO_CLUB. The set difference operator, therefore, allows us to find all members of the sport club who are not also part of the photo club.

The basic relationship between the set intersection operator and the set difference operator can be expressed as a formula:

$$R \cap S = R \setminus (R \setminus S).$$

The determination of set intersections is, therefore, based on the calculation of set differences, as can be seen in our example with the sports and photography club members.

The last remaining set operator is the Cartesian product of two arbitrary relations R and S that do not need to be union-compatible. The Cartesian product R × S of two relations R and S is the set of all possible combinations of tuples from R with tuples from S.

To illustrate this, Fig. 3.6 shows a table COMPETITION containing a combination of members of (SPORTS_CLUB\PHOTO_CLUB) × PHOTO_CLUB, i.e., all possible combinations of sports club members (who are not also members of the photo club) and photo club members. It shows a typical competition constellation for the two clubs. Of course, Howard as a member of both clubs cannot compete against themselves and enters on the photography club side due to the set difference SPORTS_CLUB\PHOTO_CLUB.

This operation is called a Cartesian product because all respective entries of the original tables are multiplied with those of the other. For two arbitrary relations R and S with m and n entries, respectively, the Cartesian product R × S has m times n tuples.

COMPETITION = (SPORTS_CLUB \ PHOTO_CLUB) × PHOTO_CLUB

E#	Name	Street	City	*E#*	Name	Street	City
E1	Murphy	Murray Road	Kent	E4	Bell	S Water Street	Kent
E1	Murphy	Murray Road	Kent	E7	Howard	Lorain Avenue	Cleveland
E19	Stewart	E Main Street	Stow	E4	Bell	S Water Street	Kent
E19	Stewart	E Main Street	Stow	E7	Howard	Lorain Avenue	Cleveland

Fig. 3.6 COMPETITION relation as an example of Cartesian products

3.2.3 Relational Operators

The relation-based operators complement the set operators. For relation-based operators, just like for the Cartesian product, the relations involved do not have to be union-compatible. A projection $\pi_a(R)$ with the project operator π forms a subrelation of the relation R based on the attribute names defined by *a.* For instance, given a relation R with the attributes (A,B,C,D), the expression $\pi_{A,C}(R)$ reduces R to the attributes A and C. The attribute names in a projection do not have to be in order; e.g., $R' := \pi_{C,A}(R)$ means a projection of R = (A,B,C,D) onto R′ = (C,A).

The first example in Fig. 3.7, π_{City}(EMPLOYEE), lists all places of residence from the EMPLOYEE table in a single-column table without any repetitions. Example two, $\pi_{Sub,Name}$(EMPLOYEE), results in a subtable with all department numbers and names of the respective employees.

The select operator σ in an expression σF(R) extracts a *selection of tuples from the relation R based on the formula F.* F consists of a number of attribute names and/or value constants connected by relational operators, such as <, >, or =, or by logical operators, e.g., AND, OR, or NOT; σF(R), therefore, includes all tuples from R that meet the selection condition F.

This is illustrated by the examples for selection of tuples from the EMPLOYEE table in Fig. 3.8: In the first example, all employees meeting the condition 'City = Kent', i.e., living in Kent, are selected. Example two with the condition 'Sub = D6' picks out only those employees working in department D6. The third and last example combines the two previous selection conditions with a logical connective, using the formula 'City = Kent AND Sub = D6'. This results in a singleton relation, since only employee Bell lives in Kent and works in department D6.

Of course, the operators of relational algebra as described above can also be combined with each other. For instance, if we first do a selection for employees of department D6 by $\sigma_{Sub=D6}$(EMPLOYEE), then project on the City attribute using the operator $\pi_{City}(\sigma_{Sub=D6}(EMPLOYEE))$, we obtain a result table with the two towns of Stow and Kent.

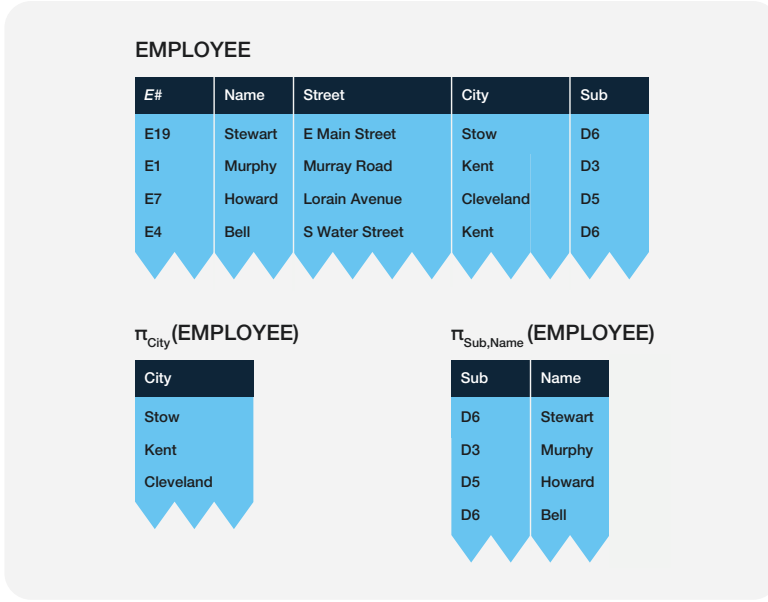


Fig. 3.7 Sample projection on EMPLOYEE

Fig. 3.8 Examples of selection operations

Next is the *join operator*, which merges two relations into a single one. The join $R|\times|_P S$ of the two relations R and S by the predicate P is a *combination of all tuples from R with all tuples from S where each meets the join predicate P.* The join predicate contains one attribute from R and one from S. Those two attributes are correlated by a relational operator (<, >, or =) so that the relations R and S can be combined. If the join predicate P uses the relational operator =, the result is called an *equi–join.*

The join operator often causes misunderstandings which may lead to wrong or unwanted results. This is mostly due to the predicate for the combination of the two tables being left out or ill-defined.

For example, Fig. 3.9 shows two join operations with and without a defined join predicate. By specifying EMPLOYEE $|\times|_{Sub=D\#}$DEPARTMENT, we join the EMPLOYEE and DEPARTMENT tables by expanding the employee information with their respective departments.

Should we forget to define a join predicate in the example in Fig. 3.9 and simply specify EMPLOYEE × DEPARTMENT, we obtain the Cartesian product of the two tables EMPLOYEE and DEPARTMENT. This is a rather meaningless combination of the two tables, since all employees are juxtaposed with all departments, resulting in combinations of employees with departments they are not actually part of (see also the COMPETITION table in Fig. 3.6).

As is shown by the examples in Fig. 3.9, the join operator $|\times|$ with the join predicate P is merely a limited Cartesian product.

In fact, a join of two tables R and S without a defined join predicate P expresses the Cartesian product of the R and S tables, i.e., for an empty predicate $P = \{\}$,

$$R|\times|_{P=\{\}} S = R \times S$$

Using a join predicate as the selection condition in a select operation yields

$$R|\times|_P S = \sigma_P(R \times S).$$

This general formula demonstrates that each join can be expressed using first a Cartesian product and second a selection.

Referring to the example in Fig. 3.9, we can calculate the intended join EMPLOYEE $|\times|_{Sub=D\#}$ DEPARTMENT with the following two steps: First we generate the Cartesian product of the two tables EMPLOYEE and DEPARTMENT. Then all entries of the preliminary result table meeting the join predicate Sub = D# are determined using the selection $\sigma_{Sub=D\#}$(EMPLOYEE × DEPARTMENT). This gives us the same tuples as calculating the join EMPLOYEE $|\times|_{Sub=D\#}$ DEPARTMENT directly (tuples marked in yellow in Fig. 3.9).

A *division* of the relation R by the relation S is only possible if S is contained within R as a subrelation. The divide operator R÷S calculates a subrelation R′ from R, which has the property that *all possible combinations* of the tuples r′ from R′ with the tuples s from S are part of the relation R, i.e., the Cartesian product R′ × S must be contained within R.

EMPLOYEE

E#	Name	Street	City	Sub
E19	Stewart	E Main Street	Stow	D6
E1	Murphy	Murray Road	Kent	D3
E7	Howard	Lorain Avenue	Cleveland	D5
E4	Bell	S Water Street	Kent	D6

DEPARTMENT

D#	DepartmentName
D3	IT
D5	HR
D6	Accounting

EMPLOYEE $|\times|_{Sub=D\#}$ DEPARTMENT

E#	Name	Street	City	Sub	D#	DepartmentName
E19	Stewart	E Main Street	Stow	D6	D6	Accounting
E1	Murphy	Murray Road	Kent	D3	D3	IT
E7	Howard	Lorain Avenue	Cleveland	D5	D5	HR
E4	Bell	S Water Street	Kent	D6	D6	Accounting

EMPLOYEE × DEPARTMENT

E#	Name	Street	City	Sub	*D#*	DepartmentName
E19	Stewart	E Main Street	Stow	D6	D3	IT
E19	Stewart	E Main Street	Stow	D6	D5	HR
E19	Stewart	E Main Street	Stow	D6	D6	Accounting
E1	Murphy	Murray Road	Kent	D3	D3	IT
E1	Murphy	Murray Road	Kent	D3	D5	HR
E1	Murphy	Murray Road	Kent	D3	D6	Accounting
E7	Howard	Lorain Avenue	Cleveland	D5	D3	IT
E7	Howard	Lorain Avenue	Cleveland	D5	D5	HR
E7	Howard	Lorain Avenue	Cleveland	D5	D6	Accounting
E4	Bell	S Water Street	Kent	D6	D3	IT
E4	Bell	S Water Street	Kent	D6	D5	HR
E4	Bell	S Water Street	Kent	D6	D6	Accounting

Fig. 3.9 Join of two tables with and without a join predicate

Table R in Fig. 3.10 shows which employees work on which projects. Assuming we want to determine who works on all projects from S, i.e., projects P2 and P4, we first define the table S with the project numbers P2 and P4. S is obviously contained in R, so we can calculate the division $R' := R \div S$. The result of this division is the table R′ with the two employees E1 and E4. A quick check shows that E1 and E4 do, indeed, work on both P2 and P4, since the table R contains the tuples (E1,P2), (E1,P4), (E4,P2), and (E4,P4).

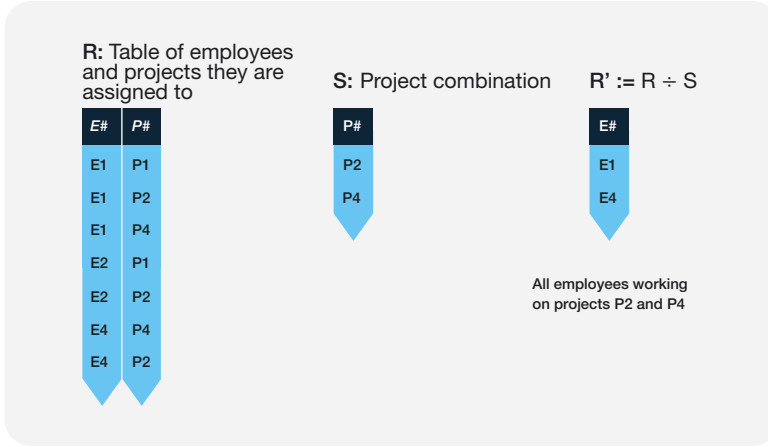


Fig. 3.10 Example of a divide operation

A divide operation can also be expressed through project and set difference operators and a Cartesian product, which makes the divide operator the third substitutable operator in relational algebra besides the set intersection and the join operator.

In summary, set union, set difference, Cartesian product, projection, and selection make up the minimal set of operators that renders relational algebra fully functional: Set intersection, join, and division can all be expressed using these five operators of relational algebra, although sometimes circuitously.

The operators of relational algebra are not only theoretically significant, but also have a firm place in practical applications. They are used in the language interfaces of relational database systems for the purpose of optimization (Sect. 5.3.2) as well as in the construction of database computers: The operators of relational algebra and their derivatives do not have to be realized in software—they can be implemented directly in hardware components.

3.3 Relationally Complete Languages

Languages are relationally complete if they are at least equivalent to relational algebra, i.e., all operations that can be executed on data with relational algebra must also be supported by relationally complete languages.

Relational algebra is the orientation point for the commonly used languages of relational database systems. We already mentioned SQL (Structured Query Language), which can be seen as a direct implementation of relational algebra (Sect. 3.3.1). QBE (Query by Example) is a language in which the actual queries and manipulations are executed via sample graphics (Sect. 3.3.2). It also supports user-friendly table handling with graphic elements.

SQL and QBE are equally as powerful as relational algebra and are, therefore, considered *relationally complete languages.* With respect to database languages, relationally complete means that they can represent the operators of relational algebra.

▶ **Completeness criterion** A database language is considered *relationally complete* if it enables at least the set operators set union, set difference, and Cartesian product, as well as the relation operators projection and selection.

This is the most important criterion for assessing a language's suitability for relational contexts. Not every language working with tables is relationally complete. If it is not possible to combine multiple tables via their shared attributes; the language is not equivalent to relational algebra and can, therefore, not be considered relationally complete.

Relational algebra is the foundation for the query part of relational database languages. Of course, it is also necessary to be able to not only analyze but also manipulate tables or individual parts. Manipulation operations include, among others, insertion, deletion, or changes to tuple sets. Relationally complete languages, therefore, need the following functions in order to be practically useful:

- It must be possible to define tables and attributes.
- Insert, change, and delete operations must be possible.
- Aggregate functions such as addition, maximum and minimum determination, or average calculation should be included.
- Formatting and printing tables by various criteria must be possible, e.g., including sorting orders and control breaks for table visualization.
- Languages for relational databases must include elements for assigning user permissions and for protecting the databases (Sect. 3.8).
- Arithmetic expressions and calculations should preferably be supported.
- Multi-user access should be supported (transaction principle, Chap. 4) and commands for data security should be included.

The definition of relational algebra has given us the formal framework for relational database languages. However, this formal language is not used in practice per se; rather, it has been a long-standing approach to try and make *relational database languages as user-friendly as possible.* Since the algebraic operators in their pure form are beyond most database users, they are represented by more accessible language elements. The following sections will give examples of SQL and QBE in order to illustrate this.

3.3.1 SQL

In the 1970s, the language SEQUEL (Structured English QUEry Language) was created for IBM's System R, one of the first working relational database systems. The concept

behind SEQUEL was to create a relationally complete query language based on *English words* such as 'select,' 'from,' 'where,' 'count,' 'group by,' etc., rather than mathematical symbols. A derivative of that language named SQL (Structured Query Language) was later standardized first by ANSI and then internationally by ISO. For years, SQL has been the leading language for database queries and interactions.

As described in Sect. 1.2.2, the basic structure of SQL looks like this:

```
SELECT selected attributes  (Output)
FROM tables to be searched (Input)
WHERE selection condition (Processing)
```

The SELECT clause corresponds to the project operator of relational algebra, in that it defines a list of attributes. In SQL, the equivalent of the project operator $\pi_{Sub,Name}$(EMPLOYEE) as shown in Fig. 3.7 is simply

```
SELECT Sub, Name
FROM EMPLOYEE
```

The FROM clause lists all tables to be used. For instance, the Cartesian product of EMPLOYEE and DEPARTMENT is expressed in SQL as

```
SELECT E#, Name, Street, City, Sub, D#, Department_Name
FROM EMPLOYEE, DEPARTMENT
```

This command generates the cross-product table in Fig. 3.9, similar to the equivalent operators EMPLOYEE$| \times |_{P=\{\}}$DEPARTMENT and EMPLOYEE $\times$ DEPARTMENT.

By setting the join predicate 'Sub = D#' in the WHERE clause, we obtain the equi-join of the EMPLOYEE and DEPARTMENT tables in SQL notation:

```
SELECT E#,Name,Street,City,Sub,D#,Department_Name
FROM EMPLOYEE, DEPARTMENT
WHERE Sub=D#
```

Qualified selections can be expressed by separate statements in the WHERE clause, connected by the logical operators AND or OR. The SQL command for the selection of employees $\sigma_{City=Kent\ AND\ Sub=D6}$(EMPLOYEE) as shown in Fig. 3.8 would be

```
SELECT *
FROM EMPLOYEE
WHERE City='Kent' AND Sub='D6'
```

An asterisk (*) in the SELECT clause means that all attributes in the table are selected, i.e., the result table contains all the attributes E#, Name, Street, City, and Sub (Subordinate). The WHERE clause contains the desired selection predicate. Executing the above query would, therefore, give us all information about the employee Bell from Kent working in department D6.

In addition to the common operators of relational algebra, SQL also contains *built-in features* that can be used in the SELECT clause. These include the aggregate functions, which calculate a scalar value based on a set, namely COUNT for counting, SUM for totaling, AVG for calculating the average, MAX for determining the maximum, and MIN for finding the minimum value.

For example, we can count all employees working in department D6. The corresponding SQL command would be

```
SELECT COUNT (E#)
FROM EMPLOYEE
WHERE Sub='D6'
```

The result is a singleton table with the single value 2, referring to the two department employees Stewart and Bell from our table excerpt.

SQL provides the CREATE TABLE command for defining a new table. The EMPLOYEE table would be specified as follows:

```
CREATE TABLE EMPLOYEE
 (E# CHAR(6) NOT NULL,
Name VARCHAR(20),
).
```

The opposite command, DROP TABLE, is used to delete table definitions. It is important to note that this command also eliminates all table contents and assigned user permissions (Sect. 3.8).

Once a table has been defined, the following command can be used to insert new tuples:

```
INSERT INTO EMPLOYEE
VALUES ('E20', 'Mahoney', 'Market Ave S', 'Canton', 'D6')
```

Existing tables can be manipulated using UPDATE statements:

```
UPDATE EMPLOYEE
SET  City = 'Cleveland'
WHERE City = 'Cuyahoga Heights'
```

This example replaces the value Cuyahoga Heights for the City attribute with the new name Cleveland in all matching tuples of the EMPLOYEE table. The UPDATE manipulation operation is set-based and can edit a multi-element set of tuples.

Entire tables or parts of tables can be eliminated with the help of DELETE statements:

```
DELETE FROM EMPLOYEE
WHERE City = 'Cleveland'
```

DELETE statements usually affect sets of tuples if the selection predicate applies to multiple entries in the table. Where referential integrity (Sect. 3.7) is concerned, deletions can also impact dependent tables.

A tutorial for SQL can be found on the website accompanying this book, www.sql-nosql.org. The short introduction given here covers only a small part of the existing standards; modern SQL offers many extensions, e.g., for programming, security, object-orientation, and analysis.

3.3.2 QBE

The language Query by Example, acronym QBE, is database language that allows users to create and execute their analysis needs directly within the table using interactive examples.

For instance, users are provided a graphic visualization of the EMPLOYEE table and the attributes it contains:

```
EMPLOYEE | E# | Name | Street | City | Sub
-------------------------------------------
         |    |      |        |      |.
```

They can use this outline for selections by inserting display commands (P.), variables, or constants. To list the employee names and the department numbers, the outline must be filled in as follows:

```
EMPLOYEE | E# | Name | Street | City | Sub
-------------------------------------------
         |    | P.   |        |      | P.
```

The "P." (short for print) command orders all data values of the respective column to be displayed. It allows projections on individual attributes of a table.

If the user wants to have all attributes in the table selected, they can enter the display command directly under the table name:

```
EMPLOYEE | E# | Name | Street | City | Sub
-------------------------------------------
P.       |    |      |        |      |.
```

Queries with added selection conditions can also be expressed, e.g., if the user wants to retrieve the names of all employees living in Kent and working in department D6. To do so, they enter the constant Kent in the City column and D6 in the Sub column:

```
EMPLOYEE | E# | Name | Street | City | Sub
-----------------------------------------------
         |    | P.   |        |'Kent'|'D6'
```

Entering selection conditions in the same row always equals an AND connective. For an OR connection, two lines are needed, as shown below:

```
EMPLOYEE | E# | Name | Street | City | Sub
-----------------------------------------------
         |    | P.   |        |'Kent'|
         |    | P.   |        |      |'D6'
```

This query is equivalent to a selection of all employees who either live in Kent or work in department D6, i.e., the result table can also be determined with the following expression from relational algebra:

$$\pi_{\text{Name}}\left(\sigma_{\text{City}=\text{Kent OR Sub}=\text{D}_6}(\text{EMPLOYEE})\right)$$

QBE queries can not only use constants, but also variables. The latter are always introduced with an underscore (_), followed by a string of characters. Variables are needed for join operations, among other purposes. If the user wants to obtain the names and addresses of the employees in the IT department, the QBE query looks like this:

```
EMPLOYEE | E# | Name | Street | City | Sub
-----------------------------------------------
         |    | P.   | P.     | P.   |_D
DEPARTMENT | D# | Department_Name
-----------------------------------------------
           | _D | 'IT'
```

The join of the two tables EMPLOYEE and DEPARTMENT is formed via the _D variable, which represents the previously used join predicate Sub = D#.

In order to add a new employee (E2, Kelly, Market Ave S, Canton, D6) to the EMPLOYEE table, the user enters the insertion command “I.” in the column with the table name and fill in the table row with the data to be entered:

```
 EMPLOYEE | E# | Name | Street         | City     | Sub
 ------------------------------------------------------
I.         |'E2'|'Kelly'|'Market Ave S'|'Canton' |'A6'
```

Users can edit or remove a set of tuples by entering the "U." (update) or "D." (delete) commands in the respective tables or columns.

For example, entries of people living in Kent can be removed from the EMPLOYEE table as follows:

```
 EMPLOYEE | E# | Name | Street | City | Sub
 ------------------------------------------------
 D.        |    |      |        |'Kent'|
```

As indicated by the name, Query by Example, unlike SQL, cannot be used to define tables or assign and manage user permissions. QBE is limited to the query and manipulation part, but is relationally complete, just like SQL.

The use of QBE in practice has shown that QBE instructions for complex queries are generally harder to read than equivalent statements in SQL, which is why especially analysis specialists prefer SQL for their sophisticated queries.

Although the QBE syntax presented here is an almost archaic example, the basic principle of querying database tables by entering parameters directly into a graphic representation of the table is still relevant today. Desktop databases for office automation (both commercial and open source solutions) as well as some database clients offer GUIs that provide end users with a QBE interface in which they can query data according to the principles described above. An example can be found in the Travelblitz case study with OpenOffice Base available on the website accompanying this book, www.sql-nosql.org.

3.4 Graph-based Languages

Graph-based database languages were first developed towards the end of the 1980s. The interest in high-performance *graph query languages* has grown with the rise of the internet and social media, which produce more and more graph-structured data.

Graph databases (Sect. 7.6) store data in graph structures and provides options for data manipulation on a graph transformation level. As described in Sect. 1.4.1, graph databases consist of property graphs with nodes and edges, with each graph storing a set of key-value pairs as properties. Graph-based database languages build on that principle and enable the use of a computer language to interact with graph structures in databases and program the processing of those structures.

Like relational languages, graph-based languages are set-based. They work with graphs, which can be defined as sets of vertices and edges or paths. Graph-based languages allow for filtering data by predicates, similar to relational languages; this filtering

is called a *conjunctive query.* Filtering a graph returns a subset of nodes and/or edges of the graph, which form a partial graph. The underlying principle is called subgraph matching, the task of finding a partial graph matching certain specifications within a graph. Graph-based languages also offer features for aggregating sets of nodes in the graph into scalar values, e.g., counts, sums, or minimums.

Unlike relational languages, graph-based languages offer additional analysis mechanisms for paths within graphs. An area of special interest is the search for patterns directly in the paths of a graph, which can be done with dedicated language elements. A *regular path query* allows to describe path patterns in a graph with regular expressions in order to find matching records in the database (see the Cypher tutorial on www.sql-nosql.org for more information).

Figure 3.11 illustrates this using an entity-relationship model of item parts. It shows a recursive relationship, where parts (e.g., product parts) can potentially have multiple subparts and at the same time also potentially be a subpart to another, superordinate part. If we want to query all subparts contained in a part *both directly and indirectly,* a simple join is not sufficient. We have to recursively go through all subparts of subparts, etc., in order to obtain a complete list.

For a long time, this kind of query could not even be defined in SQL. Only with the SQL:1999 standard did recursive queries become possible via common table expressions (CTEs); however; their formulation is still highly complicated. Defining the query for all direct and indirect subparts with a (recursive) SQL statement is rather cumbersome:

```
with recursive
rpath (partID, hasPartId, length) - CTE definition
as (
select partID, hasPartId, 1 - Initialization
from part
union all
select r.partID, p.hasPartId, r.length+1
from part p
join rpath r -- Recursive join of CTE
on (r.hasPartId = p.partID)
)
```

Fig. 3.11 Recursive relationship as entity-relationship model and as graph with node and edge types

```
select
distinct path.partID, path.hasPartId, path.length
from path -- Selection via recursive defined CTE
```

This query returns a list of all subparts for a part, plus the degree of nesting, i.e., the length of the path within the tree from the part to any (potentially indirect) subpart.

A regular path query in a graph-based language allows for simplified filtering of path patterns with regular expressions. For instance, the regular expression HAS* using a Kleene star (*) defines the set of all possible concatenations of connections with the edge type HAS (called the Kleene hull). This makes defining a query for all indirectly connected vertices in a graph-based language much easier. The example below uses the graph-based language Cypher to declare the same query for all direct and indirect subparts as the SQL example above, but in only two lines:

```
MATCH path = (p:Part) <-[:HAS*]- (has:Part)
RETURN p.partID, has.partID, LENGTH(path)
```

To summarize, the main advantage of graph-based languages is the strong alignment of their linguistic constructs on graphs, which allows for a significantly more direct linguistic definition of the processing of graph-structured data. The following section will provide more detailed information on the graph-based language Cypher as a specific example.

3.4.1 Cypher

The graph database Neo4J[2] (see also the Cypher tutorial and Travelblitz case study with Neo4J on www.sql-nosql.org) uses the language Cypher to support a language interface for the scripting of database interactions. Cypher is based on a *pattern matching* mechanism.

Similar to SQL, Cypher has language commands for data queries and data manipulation (*data manipulation language,* DML); however, the schema definition in Cypher is done implicitly, i.e., node and edge types are defined by inserting instances of them into the database as actual specific nodes and edges.

The data definition language (DDL) of Cypher can only describe indexes, unique constraints (Sect. 3.7), and statistics. Cypher does not include any direct linguistic elements for security mechanisms, for which relational languages have statements, such as GRANT and REVOKE (Sect. 3.8).

[2] http://neo4j.com.

Below, we will take a closer look at the Cypher language; all examples refer to the Northwind Data Set[3].

As described in Sect. 1.4.2, Cypher has three basic commands:

- MATCH for defining search patterns
- WHERE for conditions to filter the results
- RETURN for outputting properties, vertices, relationships, or paths.

The RETURN clause can output either vertices or property tables. The following examples returns the node with the product name Chocolade:

```
MATCH (p:Product)
WHERE p.productName = 'Chocolade'
RETURN p
```

The return of entire nodes is similar to 'SELECT *' in SQL. Cypher can also return properties as attribute values of nodes and edges in the form of tables:

```
MATCH (p:Product)
WHERE p.unitPrice > 55
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice
```

This query includes a selection, a projection, and a sorting. The MATCH clause defines a pattern matching filtering the graph for the node of the 'Product' type; the WHERE clause selects all products with a price greater than 55; and the RETURN clause projects those nodes on the properties product name and price, with the ORDER BY clause sorting the products by price.

The Cartesian product of two nodes can be generated in Cypher with the following syntax:

```
MATCH (p:Product), (c:Category)
RETURN p.productName, c.categoryName
```

This command lists all possible combination of product names and category names. However, joins of nodes, i.e., selections on the Cartesian product, are executed graph-based by matching path patterns by edge types:

```
MATCH (p:Product) -[:PART_OF]-> (c:Category)
RETURN p.productName, c.categoryName
```

[3] http://neo4j.com/developer/guide-importing-data-and-etl/.

For each product, this query lists the category it belongs to, by only considering those product and category nodes connected by edges of the PART_OF type. This equals the inner join of the 'Product' node type with the 'Category' node type via the edge type PART_OF.

There are node types where only a subset of the nodes has an edge of a specific edge type. For instance, not every employee has subordinates, i.e., only a subset of the nodes of the 'Employee' type has an incoming REPORTS_TO type edge.

So, what if the user wants to generate a list of all employees along with the number of subordinates for each employee, even if that number is zero? A MATCH (e:Employee)<-[:REPORTS_TO]-(sub) query would only return those employees who actually have subordinates, i.e., where the number is greater than zero:

```
MATCH (e:Employee) <-[:REPORTS_TO]-(sub)
RETURN e.employeeID, count(sub.employeeID)
```

An OPTIONAL MATCH clause allows to list all employees including those without subordinates:

```
MATCH (e:Employee)
OPTIONAL MATCH (e)<-[:REPORTS_TO]-(sub)
RETURN e.employeeID, count(sub.employeeID)
```

Like SQL, Cypher has operators, embedded features, and aggregate functions. The following query returns the full first name and the last name initial for each employee, along with the number of subordinates:

```
MATCH (e:Employee)
OPTIONAL MATCH (e)<-[:REPORTS_TO]-(sub)
RETURN
 e.firstName + " "
 + left(e.lastName, 1) +"." as name,
 count(sub.employeeID)
```

The operator + can be used on 'text' type data values to string them together. The embedded features LEFT returns the first n characters of a text. Finally, the aggregate 'Count' determines the number of nodes for a data value combination in the RETURN statement that exist in the set from the MATCH statement.

```
MATCH (e:Employee)
OPTIONAL MATCH (e)<-[:REPORTS_TO]-(sub)
RETURN
 e.firstName +" "
 + left(e.lastName, 1) +"." as name,
 collect(sub.employeeID)
```

Unlike in SQL, aggregates in Cypher do not require a GROUP BY clause. Additional aggregates, similar to SQL, are total (sum), minimum (min), and maximum (max). A useful nonatomic aggregate is 'collect,' which generates an array from existing values. The expression above, for example, lists all employees with their first name and their last name initial, as well as a list of the employee numbers of their subordinates (which may be empty).

Schema definition in Cypher, unlike in SQL, is done implicitly, i.e., abstract data classes (metadata) such as node and edge types or attributes are created by using them in the insertion of concrete data values. The following example inserts new data into the database:

```
CREATE
 (p:Product {
 productName:'SQL & NoSQL Databases',
 year:2016})
 -[:PUBLISHER]->
 (o:Organization {
 name:'SpringerVieweg'})
```

This expression warrants a deeper analysis because multiple things happen implicitly here. Two new nodes are created and connected, one for the product SQL and NoSQL Databases and one for the publisher Springer Vieweg. This involves the implicit generation of the new node type "Organization" that did not exist before. Those nodes are given data values in the form of attribute-value pairs entered into the nodes. Both the attributes "year" and "name" did not exist before and are, therefore, added to the schema implicitly; in SQL, this would require a CREATE TABLE and an ALTER TABLE command. Additionally, an edge with the type "PUBLISHER" is created between the nodes of the book and of the publisher, adding not only the edge itself, but also that edge type to the database schema.

SET clauses are used to change data values matching a specific pattern. The expression in the following example sets a new price for the product "Chocolade":

```
MATCH (p:Product)
WHERE p.productName = 'Chocolade'
SET p.unitPrice = 13.75
```

With DELETE, it is possible to eliminate nodes and edges as specified. Since graph databases ensure referential integrity (Sect. 3.7), vertices can only be deleted if they have no edges attached. Before being able to remove a node, the user, therefore, has to delete all incoming and outgoing edges.

Below is an expression that first recognizes all edges connected to the product "Tunnbröd," then eliminates those edges, and finally deletes the node of the product itself.

```
MATCH
 ()-[r1]->(p:Product),
 (p)-[r2]->()
WHERE p.productName = 'Tunnbröd'
DELETE r1, r2, p
```

In addition to the data manipulation we know from SQL, Cypher also supports operations on paths within the graph. In the following example, an edge of the type BASKET is generated for all product pairs that have been ordered together. This edge shows that those two products have been included in at least one order together. Once that is done, the shortest connection between any two products through shared orders can be determined with a *shortestPath* function:

```
MATCH
 (p1:Product)<-(o:Order)->(p2:Product)
CREATE
 p1-[:BASKET{order:o.orderID}]->p2,
 p2-[:BASKET{order:o.orderID}]->p1;
MATCH path =
 shortestPath(
 (p1:Product)-[b:BASKET*]->(p2:Product))
RETURN
 p1.productName, p2.productName, LENGTH(path),
 EXTRACT(r in RELATIONSHIPS(path)| r.order)
```

In addition to the names of the two products, the RETURN clause also contains the length of the shortest path between them and a list of the order numbers indirectly connecting them.

It should be noted here that, while Cypher offers some functions for analyzing paths within graphs (including the Kleene hull for edge types), it does not support the full range of Kleene algebra for paths in graphs, as required in the theory of graph-based languages. Nevertheless, Cypher is a language well suited for practical use.

3.5 Embedded Languages

Relational query and manipulation languages can not only be used interactively as independent languages, but also be embedded in an actual programming language (host language). However, embedding a relational language in a programming environment requires some provisions, which are discussed in this section.

The concept of embedded languages will be explained using SQL as an example. In order for a program to be able to read a table using a SELECT statement, it is necessary that it can pass *from one tuple to the next,* which requires a cursor concept.

3.5.1 Cursor Concept

A CURSOR is *a pointer* that can go through a set of tuples in a sequence set by the database system. Since conventional programs cannot process an entire table in one step, cursors enable a row-by-row procedure. For the selection of a table, a CURSOR must be defined in the program as follows:

```
DECLARE cursor-name CURSOR FOR <SELECT-statement>
```

This allows us to process the individual records in a table, i.e., tuple by tuple. If necessary, it is also possible to modify some or all data values of the current tuple. If the table has to be processed in a specific sequence, the above declaration must be amended by an ORDER BY clause.

Multiple CURSORs can be used within one program for navigation reasons. They have to be declared and then activated and deactivated by OPEN and CLOSE commands. The actual access to a table and the transmission of data values into the corresponding program variables happens via a FETCH command. The tapes of the variables addressed in the programming language must match the formats of the respective table fields. The FETCH command is phrased as

```
FETCH cursor-name INTO host-variable {,host-variable}
```

Each FETCH statement moves the CURSOR forward by one tuple, either according to the physical structure of the table or following the ORDER BY clause where given. If no further tuples are found, a corresponding status code is returned to the program.

Cursor concepts allow the *embedding of set-oriented query and manipulation languages into a procedural host language.* For instance, the same linguistic constructs in SQL can be used either interactively (ad hoc) or embedded (via a programming language). This has additional advantages for testing embedded programming sections, since the test tables can be analyzed and checked with interactive SQL at any point.

3.5.2 Stored Procedures and Stored Functions

From SQL:1999 onwards, SQL standards offers the possibility to embed SQL in internal database procedures and functions. Since those are stored in the data dictionary on the database server, they are called stored procedures or, if they return values, stored functions. Such linguistic elements enable the procedural processing of record sets via CURSORs and the use of branches and loops. The procedural linguistic elements of SQL were only standardized long after the language's introduction, so many vendors developed separate proprietary formats. Procedural programming with SQL is, therefore, largely product-specific.

The following example of a stored function calculates first quartile[4] of all employee salaries:

```
CREATE FUNCTION SalaryQuartile()
RETURNS INTEGER DETERMINISTIC
BEGIN
 DECLARE cnt int;
 DECLARE i int;
 DECLARE tmpSalary int;
 DECLARE employeeCursor CURSOR FOR
 SELECT Salary
 FROM Employee
 ORDER BY Salary ASC;
 SELECT COUNT(*)/4 INTO cnt FROM Employee;
 SET i := 0;
 OPEN employeeCursor;
 employeeLoop: LOOP
 FETCH employeeCursor INTO tmpSalary;
 SET i := i + 1;
 IF i >= cnt THEN
 LEAVE employeeLoop;
 END IF;
 END LOOP;
 RETURN tmpSalary;
```

This function opens a cursor on the employee table sorted by salary (low to high), loops through each row, and returns the value of the Salary column from the row where COUNT(*)/4 iterations of the loop have been run. This value is the first quartile, i.e., the value separating the lowest 25% of values in the set. The result of the function can then be selected with the statement.

```
Select SalaryQuartile();
```

3.5.3 JDBC

SQL can also be embedded in Java. Similarly to the cursor concept, Java offers the *ResultSet* class, which works as a pointer to the elements of a result set and enables the iterative processing of records. The *Java Database Connectivity (JDBC) Standard* achieved a unified interface between the Java language and a wide variety of SQL-based databases. Most SQL databases support JDBC and offer the necessary driver libraries.

[4]Quartiles of ranked data sets are the points between the quarters of the set.

The example below shows a Java class which imports data from an SQL database using an SQL expression and then processes the data:

```
Public class HelloJDBC {
 public static void main(String[] args) {
 try {
 Class.forName("com.mysql.jdbc.Driver");
 Connection connection =
 DriverManager.getConnection(
"jdbc:mysql://127.0.0.1:3306/ma",
"root",
"");
 Statement statement =
 connection.createStatement();
 ResultSet resultset =
 statement.executeQuery(
"SELECT * FROM EMPLOYEE");
 while (resultset.next()) {
 System.out.println(
 resultset.getString("Name"));
 }
 } catch (Exception e) {}
 }
}
```

This simple code snippet passes the names of all employees on to the console. However, it would also be possible to further process the data in multiple ways using SQL within the host language Java. Four main classes are especially relevant here: Connection, DriverManager, Statement, and ResultSet. Objects of the *Connection* class form a connection to the database. They are instantiated by the *DriverManager* class via the static method *getConnection,* where the access codes are also defined. The various SQL expressions can be embedded into objects of the *Statement* class. The method *executeQuery* of the *Connection* class then returns a *ResultSet* object containing a set of records as the result of the executed SQL statement. Like a CURSOR, objects of the *ResultSet* class allow the iterative, record-based processing of the result tuples.

3.5.4 Embedding Graph-based Languages

All previous examples of embedded database languages used SQL; however, since graph-based languages are also set-based, they can be embedded into host languages following the same concept using a cursor concept.

The following example shows a Java application using embedded Cypher code to access data in a Neo4J database and returns all product names to the screen.

```
try (
 Transaction t = db.beginTx();
 Result result = db.execute(
"MATCH p:Product RETURN p.productName"))
{
 while (result.hasNext()) {
 Map<String,Object> node = result.next();
 for (Entry<String,Object> property:
 node.entrySet()){
 System.out.println(property.getValue());
 }
 }
}
```

As with JDBC, the execution of an embedded Cypher statement returns a result set, which can then be processed further via a loop.

3.6 Handling NULL Values

The work with databases regularly entails situations where individual data values for tables are not (yet) known. For instance, it may be necessary to enter a new employee in the EMPLOYEE table before their full address is available. In such cases, instead of entering meaningless or maybe even wrong filler values, it is advisable to use NULL values as placeholders.

▶ **NULL values** A NULL value represents an as yet unknown data value within a table column.

NULL values, represented as "?", must not be confused with the number 0 (zero) or the value "Blank" (space). These two values express specific situations in relational databases, while NULL values are merely placeholders (unknown).

Figure 3.12 shows the EMPLOYEE table with NULL values for the attributes Street and City. Of course, not all attribute categories may contain NULL values, otherwise conflicts are unavoidable. Primary keys must not contain NULL values by definition; in our example, that applies to the employee number E#. For the foreign key "Sub", the database architect can make that decision at their discretion and based on their practical experience.

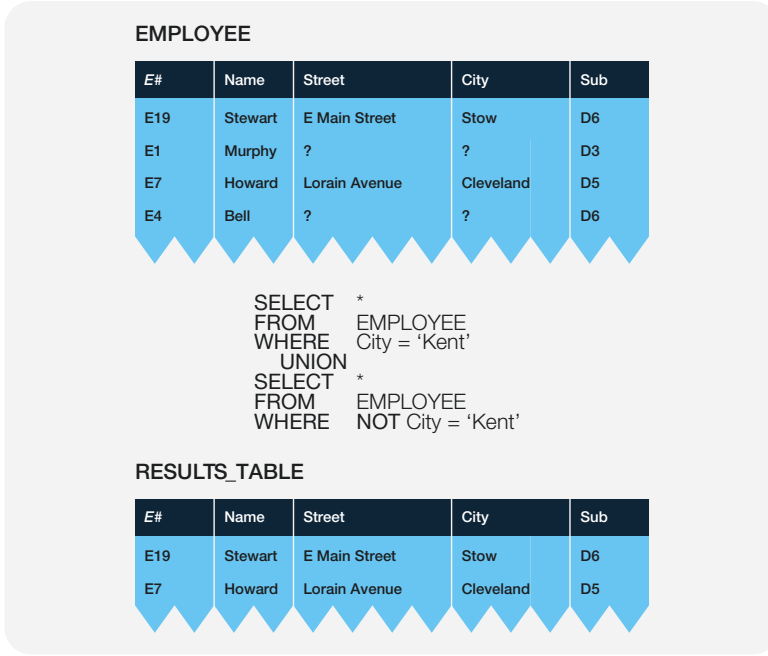
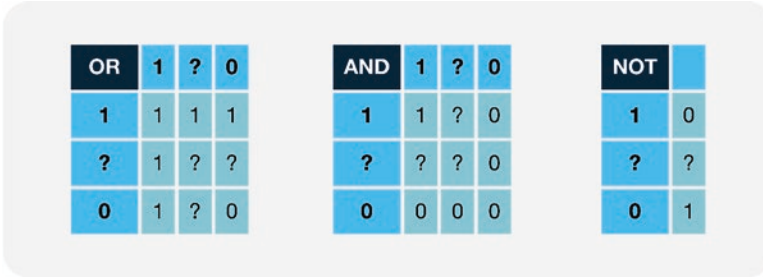


Fig. 3.12 Unexpected results from working with NULL values

Working with NULL values can be somewhat problematic, since they form a new information content UNKNOWN (?) in addition to TRUE (1) and FALSE (0). We, therefore, have to leave behind the classical binary logic in which any statement is either true or false. Truth tables for logical connectives such as AND, OR, or NOT can also be derived for three truth values. As shown in Fig. 3.13, combinations including NULL values again return NULL values, which may lead to counter-intuitive results, as in the example in Fig. 3.12.

The query in Fig. 3.12, which selects all employees from the EMPLOYEE table who either live in Kent or not in Kent, returns a result table containing only a subset of the employees in the original table, since the places of residence of some employees are *unknown* (and, therefore, not *true)*. This clearly goes against the conventional logical assumption that a union of the subset "employees living in Kent" with its complement "employees NOT living in Kent" should result in the total set of all employees.

Sentential logic with the values TRUE, FALSE, and UNKNOWN is commonly called three-valued logic for the three truth values a statement can take. This logic is less known and poses a special challenge for users of relational databases, since analyses of tables with NULL values are hard to interpret. In practice, NULL values are, therefore, largely avoided. Sometimes, DEFAULT values are used instead. For instance, the company address could be used to replace the yet unknown private addresses in the



OR	1	?	0
1	1	1	1
?	1	?	?
0	1	?	0

AND	1	?	0
1	1	?	0
?	?	?	0
0	0	0	0

NOT	
1	0
?	?
0	1

Fig. 3.13 Truth tables for three-valued logic

EMPLOYEE table from our example. The function COALESCE (X, Y) replaces all X attributes with a NULL value with the value Y. If NULL values have to be allowed, attributes can be checked for unknown values with specific relation operators, IS NULL or IS NOT NULL, in order to avoid unexpected side effects.

Foreign keys are usually not supposed to take NULL values; however, there is an exception for foreign keys under a certain rule of referential integrity. For instance, the deletion rule for the referenced table DEPARTMENT can specify whether or not existing foreign key references should be set to NULL. The referential integrity constraint "set NULL" declares that foreign key values are set to NULL if their referenced tuple is deleted. For example, deleting the tuple (D6, Accounting) from the DEPARTMENT table in Fig. 3.12 with the integrity rule "set NULL" results in NULL values for the foreign keys of employees Stewart and Bell in the EMPLOYEE table. This constraint complements the rules for restricted and cascading delete described in Sect. 2.3.3. For more information, see also Sect. 3.7.

Graph-based languages can also have NULL values and corresponding functions. Cypher is also based on three-valued logic and can handle NULL values with IS NULL and COALESCE, similar to SQL.

3.7 Integrity Constraints

The integrity of a database is a vital characteristic that must be supported by the DBMS. The respective rules applying to all insert or update operations are called *integrity constraints*. For reasons of efficiency, those rules are not specified individually in each program, but rather overall in the database schema. Integrity constraints are divided into declarative and procedural rules.

Declarative integrity constraints are defined during the generation of a new table in the CREATE TABLE statement using the data definition language. In the example in Fig. 3.14, the primary key for the DEPARTMENT table is specified as an integrity constraint with PRIMARY KEY. Primary and foreign key of the EMPLOYEE table are defined similarly.

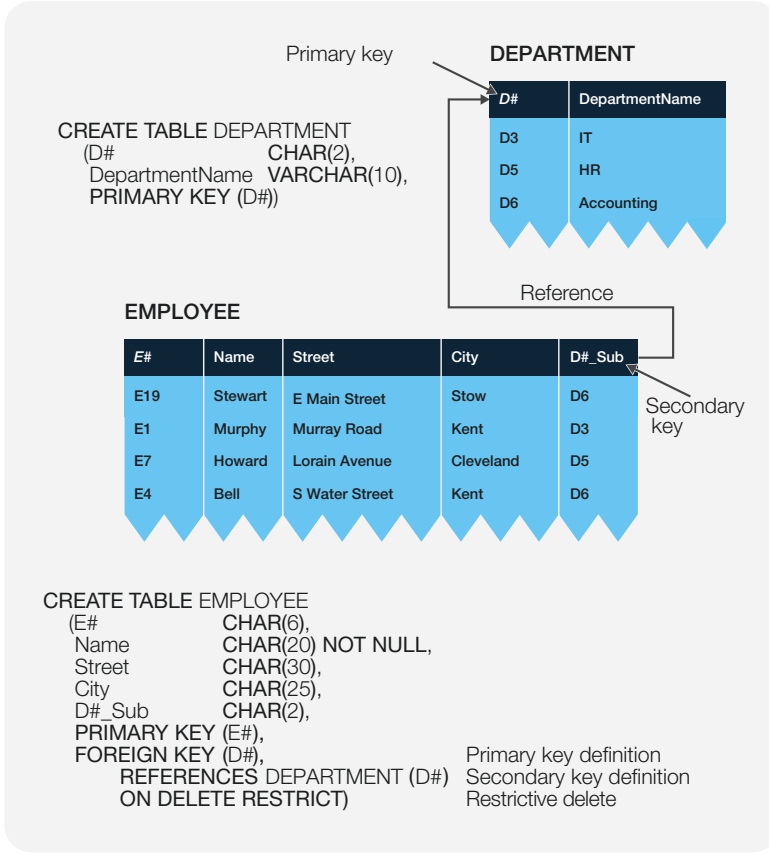


Fig. 3.14 Definition of declarative integrity constraints

The various types of declarative integrity constraints are:

- **Primary key definition**: PRIMARY KEY defines a unique primary key for a table. Primary keys must, by definition, not contain any NULL values.
- **Foreign key definition**: FOREIGN KEY can be used to specify a foreign key, which relates to another table in the REFERENCES clause.
- **Uniqueness**: The uniqueness of an attribute can be determined by the UNIQUE constraint. Unlike primary keys, unique attributes may contain NULL values.
- **No NULL values**: The NOT NULL constraint dictates that the respective attribute must not contain any NULL values. For instance, the attribute Name in the EMPLOYEE table in Fig. 3.14 is set to NOT NULL, because there must be a name for every employee.
- **Check constraint**: Such rules can be declared with the CHECK command and apply to every tuple in the table. For example, the CHECK Salary > 30.000 statement in the

STAFF table in Fig. 3.14 ensures that the annual salary of each employee is at least USD 30,000.

- **Set to NULL for changes or deletions**: ON UPDATE SET NULL or ON DELETE SET NULL declare for dependent tables that the foreign key value of a dependent tuple is set to NULL when the corresponding tuple in the referenced table is modified or removed (Sect. 2.3.3).
- **Restricted changes or deletion**: If ON UPDATE RESTRICT or ON DELETE RESTRICT is set, tuples cannot be manipulated or deleted while there are still dependent tuples referencing them (Sect. 2.3.3).
- **Cascading changes or deletion**: ON UPDATE CASCADE or ON DELETE CASCADE define that the modification or removal of a reference tuple is extended to all dependent tuples (Sect. 2.3.3).

In Fig. 3.14, a restrictive deletion rule has been specified for the two tables DEPARTMENT and EMPLOYEE. This ensures that individual departments can only be removed if they have no dependent employee tuples left.

The command

```
DELETE FROM Department WHERE D# = 'D6'
```

would, therefore, return an error message, since the employees Stewart and Bell are listed under the accounting department.

Apart from delete operations, declarative integrity constraints can also affect insert and update operations. For instance, the insert operation

```
INSERT INTO EMPLOYEE
VALUES ('E20','Kelly','Market Ave S','Canton','D7')
```

will also return an error message: Department D7 is not yet listed in the referenced table DEPARTMENT, but due to the foreign key constraint, the DBMS checks whether the key D7 exists in the referenced table before the insertion.

Declarative, or static, integrity constraints can be defined during table generation (CREATE TABLE statement). On the other hand, procedural, or dynamic, integrity constraints compare database states before and after a change, i.e., they can only be checked during runtime. The triggers are an alternative to declarative integrity constraints because they initiate a sequence of procedural branches via instructions. Triggers are mostly defined by a trigger name, a database operation, and a list of subsequent actions:

```
CREATE TRIGGER NoCuts -- trigger name
BEFORE UPDATE ON Employee -- database operation
FOR EACH ROW BEGIN -- subsequent action
IF NEW.Salary < OLD.Salary
```

```
THEN set NEW.Salary = OLD.Salary
END IF;
END
```

The example above shows a situation where employees' salaries must not be cut, so before updating the EMPLOYEE table, the trigger checks whether the new salary is lower than the old one. If that is the case, the integrity constraint is violated and the new salary is reset to the original value from before the update. This is a very basic example meant to illustrate the core concept. In a production environment, the user would also be notified.

Working with triggers can be tricky, since individual triggers may prompt other triggers, which raises the issue of terminating all subsequent actions. In most commercial database systems, the simultaneous activation of multiple triggers is prohibited to ensure a clear action sequence and the proper termination of triggers.

The only integrity constraint that can be explicitly defined in the graph-based language Cypher is the (declarative) condition that the value of an attribute for a node type must be unique. For instance, the following expression determines that the product name for nodes of the Product type has to be unique:

```
CREATE CONSTRAINT ON (p:Product)
ASSERT p.productName IS UNIQUE;
```

However, the Neo4j graph database, for which Cypher offers the language interface, *implicitly checks all data for referential integrity,* i.e., neither primary nor foreign keys have to be declared explicitly. Neo4j's database management system ensures that edges refer to existing nodes in all cases. Nodes can, therefore, only be deleted if there are no edges connected to them (Sect. 3.4.1).

3.8 Data Protection Issues

Data protection is the prevention of unauthorized access to and use of data. Protective measures include procedures for the positive identification of a person or for the assignment of user permissions for specific data access as well as cryptographic methods for confidential data storage and transmission.

In contrast, *data security* means the hardware and software solutions that help to protect data from falsification, destruction, and loss. Security measures for database backup and recovery are discussed in Chap. 4.

The relational model facilitates the implementation of reliable restrictions to ensure data protection. A major data protection mechanism in relational databases is to provide users with only those tables and table sections they need for their work. This is done

with table *views*, each of which is based on either one or multiple physical tables and is defined using a SELECT statement:

```
CREATE VIEW view-name AS<SELECT-statement>
```

Figure 3.15 shows two example views based on the STAFF table. The EMPLOYEE view shows all attributes except for the salary information. The view GROUP_A shows only those employees with their respective salaries who earn between USD 80,000 and 100,000 p.a. Other views can be defined similarly, e.g., to allow HR to access confidential data per salary group.

The two examples in Fig. 3.15 demonstrate important protection methods. On the one hand, tables can be limited for specific user groups by projection on only certain attributes; on the other hand, access control can also be value-based, e.g., for salary ranges, via corresponding view definitions in the WHERE clause.

As in tables, it is possible to formulate queries on views; however, manipulation operations cannot always be defined uniquely. If a view is defined as a join of multiple tables, change operations may be denied by the database system under certain circumstances.

Fig. 3.15 Definition of views as part of data protection

▶ **Updateable view** Updateable views allow for insert, delete, and update operations. The following criteria determine whether a view is updateable:

- The view contains content from only one table (no joins allowed).
- That base table has a primary key.
- The defining SQL expression contains no operations that affect the number of rows in the result set (e.g., *aggregate, group by, distinct,* etc.).

It is important to note that different views of a single table with the included data are not managed redundantly to the base table; rather, merely the definitions of the views are stored. Only when the view is queried with a SELECT statement are the corresponding result tables generated from the view's base tables with the permitted data values.

Effective data protection takes more than just limiting access to table contents by views. It also has to be possible to set user specific authorizations for table functions. In SQL, user privileges can be managed with the commands GRANT and REVOKE.

Permissions are assigned with GRANT and can be removed with REVOKE.

```
GRANT <privilege> ON <table> TO <user>
REVOKE <privilege> ON <table> FROM <user>
```

GRANT changes the privilege list so that the affected users are permitted to execute read, insert, or delete operations on certain tables or views. Such granted authorizations can be taken back with REVOKE.

For example, it is possible to grant only reading privileges for the EMPLOYEE view from Fig. 3.15:

```
GRANT SELECT ON EMPLOYEE TO PUBLIC
```

Instead of listing specific users, this example uses PUBLIC to assign reading privileges to all users so they can look at the limited EMPLOYEE view of the base table.

For a more selective assignment of permissions, individual user can be specified. For instance, it is possible to authorize only a certain HR employee with the user ID ID37.289 to make changes to the GROUP_A view from Fig. 3.15:

```
GRANT UPDATE ON GROUP_A TO ID37.289
WITH GRANT OPTION
```

User ID37.289 can now modify the GROUP_A view and, thanks to the GRANT OPTION, even assign this authorization or a limited reading privilege to others and take it back. This concept allows us to define and manage dependencies between privileges.

Unlike SQL, *Cypher does not offer any linguistic elements for permission management* on the level of abstract data types (node or edge types). Neo4j explicitly does not support access constraints on the data level. While there is user authentication involving

usernames and passwords, access is granted to each user for the entire database. Security mechanisms can only be set on the network level for individual database servers. It is also possible to program specific authorization rules for Neo4j servers in Java, but this requires specific programming knowledge and is nowhere near as user-friendly as the linguistic security mechanisms of SQL.

The complexity of managing the assignment and removal of permissions when giving end users access to a relational query and manipulation language is not to be underestimated, even if the data administrators can use GRANT and REVOKE commands. In reality, daily changes and the monitoring of user authorizations require additional management tools. Internal or external controlling instances and authorities may also demand special measures to constantly ensure the proper handling of especially sensitive data (see also the legal data protection obligations for your jurisdiction).

SQL injection

A major aspect for the security of databases connected to the internet is the prevention of *SQL injections*. Since websites are often coded and connected to a database on the server side, the respective server scripts usually generate SQL code to create an interface with the database (Sect. 3.5). Where this generated SQL code contains parameters that can be modified by users (e.g., in forms or as part of URLs), it is possible that sensitive data from the database is exposed or manipulated.

To illustrate, imagine an online store that displays the customer's stored payment methods after log-in. The page showing the payment methods has the URL

```
http://example.net/payment?uid=117
```

A Java servlet runs in the background, getting the credit card information (card number and name) from the database for the page to display as a table with HTML.

```
ResultSet resultset =
 statement.executeQuery(
"SELECT creditcardnumber, name FROM PAYMENT" +
"WHERE uid ="+ request.getParameter("uid"));
while (resultset.next()) {
 out.println("<tr><td>" +
 resultset.getString("creditcardnumber") +
"</td> <td>" +
 resultset.getString("name") +"</td></tr>"
}
```

This involves the dynamic generation of an SQL query on the PAYMENT table using the customer's identification number (uid) as the selection parameter. This code generation is vulnerable to SQL injection. By extending the uid parameter in the URL as shown below, users can access the credit card information of all customers:

```
http://example.net/payment?uid=117%20OR%201=1
```

With this GET parameter, the servlet described above generates the following SQL code:

```
SELECT creditcardnumber, name
FROM PAYMENT
WHERE uid = 117 OR 1=1;
```

The additional SQL code "OR 1 = 1", the *SQL injection,* effectively disables the user identification filter in the generated query, since 1 = 1 is always correct and an OR statement is true even if only one of the conditions is true. The webpage in this simple example would, therefore, expose highly sensitive information with the SQL injection.

SQL injections are a substantial security issue, and hackers have repeatedly managed to breach even high-profile websites with this technique. There are several options to protect websites from SQL injections: On the one hand, it is becoming increasingly common to use NoSQL databases such as MongoDB or CouchDB in web development, where the lack of SQL interfaces naturally removes any vulnerability against SQL injections. If SQL databases are still used in web environments, the SQL code generation can be transferred to strictly standardized stored functions on the database (Sect. 3.5.2). In the example above, such a function could accept only a *purely numerical user ID* as input in order to return the credit card information. Injecting this function with the text "OR 1 = 1" would result in an error message.

In summary, SQL databases provide comprehensive security mechanisms based on the CREATE VIEW, GRANT, and REVOKE commands that no NoSQL database comes close to achieving. However, within the larger context of web-based information systems, these control mechanisms can be defeated by SQL injections. In such cases, NoSQL databases with their more direct APIs can offer better protection.

3.9 Further Reading

The early works of Codd (1970) describe both the relational model and relational algebra. Further deliberations on relational algebra and relational calculus can be found in leading works by Date (2004), Elmasri and Navathe (2006), and Maier (1983), while Ullman (1982) demonstrates the equivalence of the two.

SQL emerged from the research of Astrahan et al. (1976) for the relational database system System R; QBE was also developed in the 1970 s by Zloof (1977).

The database handbook of Lockemann and Schmidt (1993) gives an overview of various query and manipulation languages in German. Linguistic aspects are the focus of German works by Saake et al. (2007), Kemper and Eickler (2013), and Lang and Lockemann (1995).

There are multiple German textbooks on SQL, e.g., Beaulieu (2006), Kuhlmann and Müllmerstadt (2004), Panny and Taudes (2000), and Sauer (2002).

Darwen and Date (1997) discuss the standard of SQL; Pistor (1993) focuses on the object-oriented concepts of the SQL standard in his German article. The comprehensive work of Melton and Simon (2002) describes SQL:1999, while the German book by Türker (2003) examines both the SQL:1999 and SQL:2003 standards.

Regarding the field of graph-based languages, there are no international standards yet. Wood (2015) provides an overview of graph-based languages. The work of He and Singh (2010) describes GraphQL, a language based on a graph algebra and equally powerful as relational algebra. The only book on the commercial language Cypher so far is that by Panzarino (2014). Additional sources regarding Cypher can be found on the website of the developing company Neo4j.

References

Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D., Eswaran, K.P., Gray, J.N., Griffiths, P.P., King, W.F., Lorie, R.A., McJones, P.R., Mehl, J.W., Putzolu, G.R., Traiger, I.L., Wade, B.W., Watson, V.: System R—Relational approach to database management. ACM Trans. Database Syst. **1**(2), 97–137 (1976)

Beaulieu A.: Einführung in SQL. O'Reilly, Köln (2006)

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

Darwen H., Date C. J.: A Guide to the SQL Standard. Addison Wesley, Reading (1997)

Date C. J.: An Introduction to Database Systems. Addison-Wesley, Boston (2004)

He H., Singh A.K.: Query language and access methods for graph databases. In: Aggarwal C.C., Wang H. (eds.) Managing and Mining Graph Data. pp. 125–160. Springer, Berlin (2010)

Kemper A., Eickler A.: Datenbanksysteme—Eine Einführung. Oldenbourg, Berlin (2013)

Kuhlmann G., Müllmerstadt F.: SQL—Der Schlüssel zu relationalen Datenbanken. Rowohlt, Reinbek bei Hamburg (2004)

Lang S. M., Lockemann P. C.: Datenbankeinsatz. Springer, Berlin (1995)

Lockemann P.C., Schmidt J.W. (Hrsg.): Datenbank-Handbuch. Springer, Berlin (1993)

Maier D.: The Theory of Relational Databases. Computer Science Press, Rockville (1983)

Melton J., Simon A.R.: SQL1999—Understanding Relational Language Components. Morgan Kaufmann, San Francisco (2002)

Panny W., Taudes: Einführung in den Sprachkern SQL-99. Springer, Berlin (2000)

Panzarino, O.: Learning Cypher. Packt Publishing Ldt, Birmingham (2014)

Pistor, P.: Objektorientierung in SQL3: Standard und Entwicklungstendenzen. Informatik-Spektrum **16**(2), 89–94 (1993)

Sauer H.: Relationale Datenbanken—Theorie und Praxis inklusive SQL-2. Addison Wesley, Bonn (2002)

Türker C.: SQL:1999 & SQL:2003—Objektrelationales SQL, SQLJ & SQL/XML. dpunkt, Heidelberg (2003)

Ullman J.: Principles of Database Systems. Computer Science Press, Rockville (1982)

Wood, P.T.: Query languages for graph databases. SIGMOD Rec. **41**(1), 50–60 (2012). http://doi.org/10.1145/2206869.2206879. Accessed 24 August 2015

Zloof, M.M.: Query-by-Example: A data base language. IBM Syst. J. **16**(4), 324–343 (1977)

Ensuring Data Consistency

4

4.1 Multi-User Operation

The terms consistency and integrity of a database describe a state in which the stored data does not contradict itself. Integrity constraints (Sects. 2.4.3 and 3.7) are to ensure that data consistency is maintained for all insert and update operations.

One potential difficulty arises when multiple users simultaneously access a database and modify contained data. This can cause conflicts involving blocking each other (deadlocks) or even consistency violations. Depending on the use case, breaches of consistency rules are absolutely unacceptable. A classic example is posting transactions in banking, where the principles of double-entry bookkeeping must always be observed and must not be violated (Sect. 4.2.2).

Transaction management systems ensure that consistent database states are only changed to other consistent database states. These systems follow an all-or-none rule to prevent transactions from executing partial changes to the database. Either all requested changes are applied, or the database is not modified at all. Pessimistic or optimistic concurrency control methods are used to guarantee that the database remains in a consistent state at any time.

However, with comprehensive web applications, it has been shown that striving for full consistency is not always desirable. This is due to the CAP theorem, which states that any database can, at most, have two out of three: consistency, availability, or partition tolerance. Therefore, if the focus is on availability and partition tolerance, temporarily inconsistent database states are unavoidable.

Section 4.2 will explain the classic concept of transactions, which is based on atomicity, consistency, isolation, and durability and is known as the ACID principle for short. Section 4.3 will discuss the abovementioned CAP theorem and the light version for ensuring consistency known as BASE (basically available, soft-state, eventually

A. Meier and M. Kaufmann, *SQL & NoSQL Databases,*
https://doi.org/10.1007/978-3-658-24549-8_4

consistent), which allows for replicated computer nodes to temporarily hold different data versions and only be updated with a delay. Section 4.4 compares the ACID and BASE approaches, and Sect. 4.5 contains literature for further reading.

4.2 Transaction Concept

4.2.1 ACID

Ensuring the integrity of data is a major requirement for many database applications. The *transaction management* of a database system *allows conflict-free simultaneous work by multiple users*. Changes to the database are only applied and become visible if all integrity constraints as defined by the users are fulfilled.

The term *transaction* describes database operations bound by integrity rules, which update database states while maintaining consistency. More specifically, a transaction is a sequence of operations that has to be atomic, consistent, isolated, and durable.

- **Atomicity (A)**: Transactions are either applied in full or not at all, leaving no trace of their effects in the database. The intermediate states created by the individual operations within a transaction are not visible to other concurrent transactions. A transaction can, therefore, be seen as a *unit for the resettability* of incomplete transactions (Sect. 4.2.5).
- **Consistency (C)**: During the transaction, integrity constraints may be temporarily violated; however, at the end of the transaction, all of them must be met again. A transaction, therefore, always results in moving the database from one consistent state into another and ensures the integrity of data. It is considered a *unit for maintaining consistency*.
- **Isolation (I)**: The concept of isolation requires that parallel transactions generate the same results as transactions in single-user environments. Isolating individual transactions from transactions executed simultaneously protects them from unwanted side effects. This makes transactions a *unit for serializability*.
- **Durability (D)**: Database states must remain valid and be maintained until they are changed by a transaction. In the case of software errors, system crashes, or errors on external storage media, durability retains the effects of a correctly completed transaction. In relation to the reboot and recovery procedures of databases, transactions can be considered a unit for recovery (Sect. 4.2.5).

These four principles, atomicity (A), consistency (C), isolation (I), and durability (D), describe the *ACID concept of transactions*, which is the basis of several database systems and guarantees that all users can only make changes that lead from one consistent database state to another. Inconsistent interim states remain invisible externally and are rolled back in the case of errors.

To declare a series of operations as one transaction, they should be marked with BEGIN TRANSACTION and END_OF_TRANSACTION[1]. The start and end of a transaction indicate to the database system which operations form a unit and must be protected by the ACID concept.

The SQL statement COMMIT applies the changes from the transaction. They remain until changed by another successfully completed transaction. In the case of an error during the transaction, the entire transaction can be undone with the SQL command ROLLBACK.

The SQL standard allows for the degree of consistency enforced by the database system to be configured by setting an isolation level with the following expression:

```
SET TRANSACTION ISOLATION LEVEL <isolation level>
```

There are four isolation levels: READ UNCOMMITTED (no consistency enforcement), READ COMMITTED (only applied changes can be read by other transactions), REPEATABLE READ (read queries give the same result repeatedly), and SERIALIZABLE (full serializable ACID consistency enforced).

4.2.2 Serializability

A major aspect in the definition of operation systems and programming languages is the coordination or synchronization of active processes and the mutual exclusion of simultaneous processes. For database systems, too, concurrent accesses to the same data objects must be serialized in order for database users to be able to work independently of each other.

▶ **Concept of serializability** A system of simultaneous transactions is *synchronized correctly if there is a serial execution creating the same database state*.

The principle of serializability ensures that the results in the database are identical, whether the transactions are executed one after the other or in parallel. The focus in defining conditions for serializability is on the READ and WRITE operations within each transaction, i.e., the operations which *read and write records in the database*.

Banking provides typical examples of concurrent transactions. The basic integrity constraint for posting transactions is that debit and credit have to be balanced. Figure 4.1 shows two simultaneously running posting transactions with their READ and WRITE operations in chronological order. Neither transaction on its own changes the total

[1]In the SQL standard, transactions are implicitly started by SQL statements and concluded by COMMIT. Alternatively, they can be initiated explicitly with START TRANSACTION.

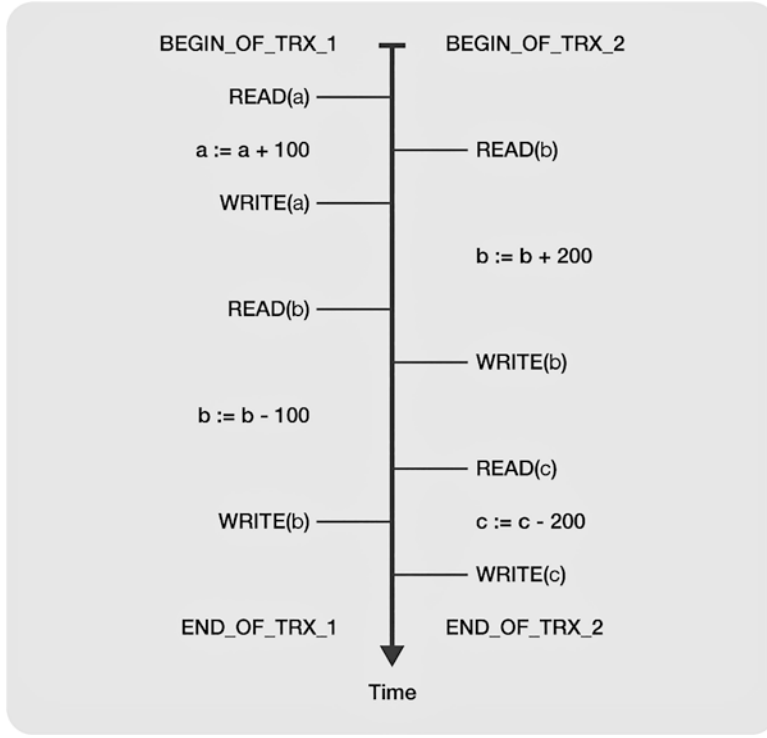


Fig. 4.1 Conflicting posting transactions

amount of the accounts a, b, and c. The transaction TRX_1 credits account a with 100 units of currency and, at the same time, debits account b with 100 units of currency. The posting transaction TRX_2 similarly credits account b and debits account c for 200 currency units each. Both transactions, therefore, fulfill the integrity constraint of bookkeeping, since the ledgers are balanced.

However, if both transactions are executed simultaneously, a *conflict* arises: The transaction TRX_1 misses the credit b:=b+200[2] done by TRX_2, since this change is not immediately written back, and reads a "wrong" value for account b. After both transactions are finished, account a holds the original amount+100 units (a+100), the amount in account b is reduced by 100 units (b−100), and c holds 200 units less (c−200). Due to the Transaction TRX_1 missing the b+200 step for account b and not calculating the amount accordingly, the total credits and debits are not balanced and the integrity constraint is violated.

[2]The notation b:=b+200 means that the current balance of account b is increased by 200 currency units.

Potential conflicts can be discovered beforehand. To do so, those READ and WRITE operations affecting a certain object, i.e., a single data value, a record, a table, or sometimes even an entire database, are filtered from all transactions. The *granularity* (relative size) of the object decides how well the transactions picked can be synchronized. The larger the granularity, the smaller the degree of transaction synchronization and vice versa. All READ and WRITE operations from different transactions that apply to a specific object are, therefore, listed in the *log* of the object x, short LOG(x). The LOG(x) of object x contains, in chronological order, all READ and WRITE operations accessing the object.

In our example of the concurrent posting transactions TRX_1 and TRX_2, the objects in question are the accounts a, b, and c. As shown in Fig. 4.2, the log for object b, for instance, contains four entries (Fig. 4.1). First, TRX_2 reads the value of b, then TRX_1 reads the same value, before TRX_2 gets to write back the modified value of b. The last log entry is caused by TRX_1 when it overwrites the value from TRX_2 with its own modified value for b. Assessing the logs is an easy way to analyze conflicts between concurring transactions. A *precedence graph* represents the transactions as nodes and possible READ_WRITE or WRITE_WRITE conflicts as directed edges (arched arrows). For any one object, WRITE operations following READs or WRITEs can lead to conflicts, while multiple READ operations are generally not a conflict risk. The precedence graph does, therefore, not include any READ_READ edges.

Figure 4.2 shows not only the log of object b for the posting transactions TRX_1 and TRX_2, but also the corresponding precedence graph. Starting from the TRX_1 node, a READ on object b is followed by a WRITE on it by TRX_2, visualized as a directed edge from the TRX_1 node to the TRX_2 node. According to the log, a WRITE_WRITE edge goes from the TRX_2 node to the TRX_1 node, since the WRITE operation by TRX_2 is succeeded by another WRITE on the same object by TRX_1. The precedence graph is, therefore, cyclical, in that there is a directed path from a node that leads back

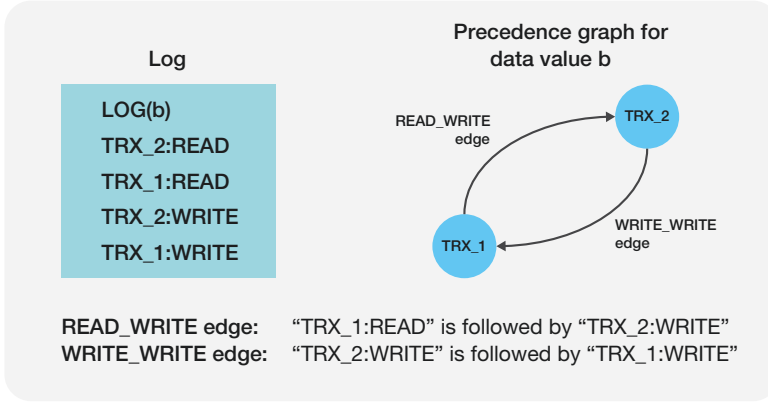


Fig. 4.2 Analyzing a log using a precedence graph

to the same node. This cyclical dependency between the transactions TRX_1 and TRX_2 shows that they are not serializable.

▶ **Serializability condition** A set of transactions is *serializable* if the corresponding precedence graphs contain *no cycles*.

The serializability condition states that multiple transactions have to yield the same results in a multi-user environment as in a single-user environment. In order to ensure serializability, *pessimistic methods* prevent any concurrent transaction runs that would lead to conflicts, while *optimistic methods* accept the chance of conflicts and fix them retroactively by rolling back the respective transactions.

4.2.3 Pessimistic Methods

Transactions can secure themselves from interferences by others by using locks to prevent additional accesses to the objects they need to read or update. *Exclusive locks* let only one transaction access the affected object, while concurring transactions that require access to the same object are rejected or queued. If such a lock is placed on an object, all other transactions that need this object have to wait until the object is released again.

The *locking protocol* defines how locks are set and released. If locks are cleared too early or without proper care, nonserializable sequences can arise. It is also necessary to prevent multiple transactions from blocking each other and creating a deadlock.

The exclusive locking of objects requires the operations LOCK and UNLOCK. Every object has to be locked before a transaction can access it. While an object x is blocked by a LOCK(x), no other transaction can read or update it. Only after the lock on object x has been released by UNLOCK(x) can another transaction place a new lock on it.

Normally, locks follow a well-defined protocol and cannot be requested or released arbitrarily.

▶ **Two-phase locking protocol** Two-phase locking (2PL) prevents a transaction from *requesting an additional LOCK after the first UNLOCK*.

Transactions under this locking protocol are always executed in two phases: During the *expanding phase*, all locks are requested and placed; during the *shrinking phase*, the locks are released one by one. This means that during the expanding phase of a transaction with 2PL, LOCKs can only be placed, gradually or all at once, but never released. UNLOCK operations are only allowed during the shrinking phase, again individually or in total at the end of the transaction. Two-phase locking effectively prohibits an intermix of creating and releasing locks.

Figure 4.3 shows a possible 2PL protocol for the posting transaction TRX_1. During the expanding phase, first account a is locked, then account b, before both accounts are

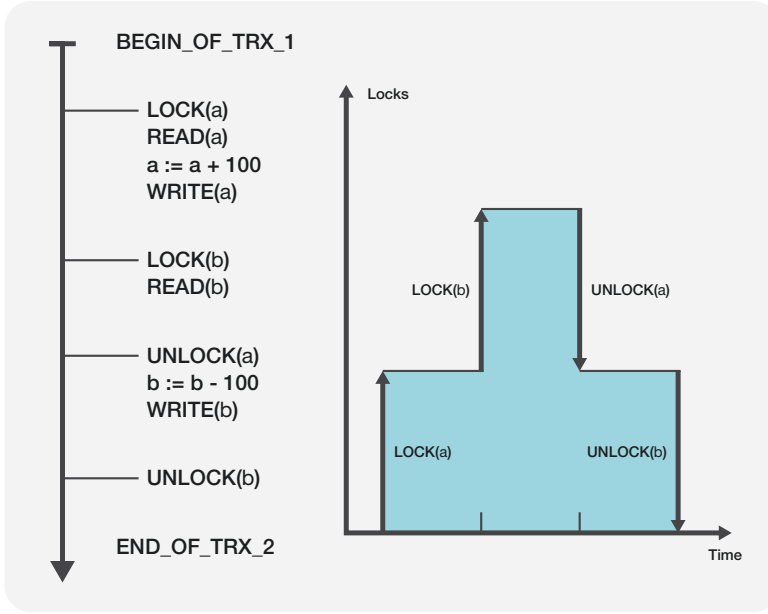


Fig. 4.3 Sample two-phase locking protocol for the transaction TRX_1

released again in the same order. It would also be possible to have both locks in this example created right at the beginning of the transaction instead of one after the other. Similarly, they could both be released at once at the end of the transaction, rather than progressively.

However, requesting the locks on the objects a and b one by one during the expanding phase and releasing them individually during the shrinking phase increases the degree of synchronization for TRX_1. If both locks were set at the beginning and only lifted at the end of the transaction, concurring transactions would have to wait the entire processing time of TRX_1 for the release of objects a and b.

Overall, two-phase locking ensures the serializability of simultaneous transactions.

Pessimistic concurrency control: With the help of two-phase locking, any set of concurring transactions is serializable. Due to the strict separation of expanding and shrinking phases, the 2PL protocol prevents any cyclical dependencies in all precedence graphs from the start; the concurring transactions remain free of conflict. In the case of the two posting transactions TRX_1 and TRX_2, this means that with properly planned locking and unlocking, they can be synchronized without any violation of the integrity constraint.

Figure 4.4 shows how such a conflict-free parallel run of TRX_1 and TRX_2 can be achieved. LOCKs and UNLOCKs are set according to 2PL rules, so that, for instance, account b is locked by TRX_2 and can only be unlocked during the transaction's shrinking phase, while TRX_1 has to wait to get its own lock on b. Once TRX_2 releases

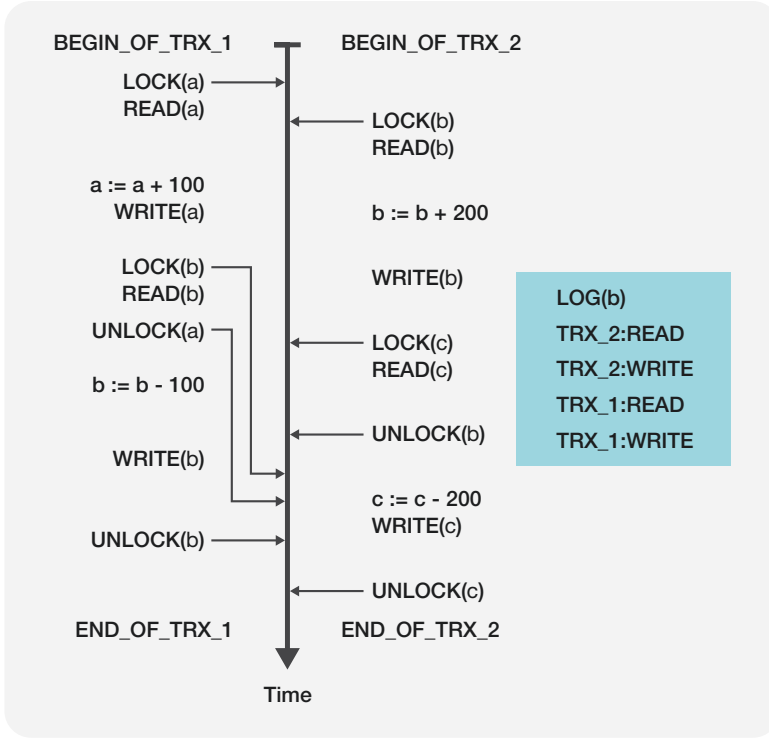


Fig. 4.4 Conflict-free posting transactions

account b via UNLOCK(b), TRX_1 requests access to and a lock on b. In this run, TRX_1 reads the correct value for b, i.e., b+200. The two transactions TRX_1 and TRX_2 can, therefore, be executed simultaneously.

2PL causes a slight delay in the transaction TRX_1, but after both transactions are finished, integrity is retained. The value of account a has increased by 100 units (a+100), as has the value of account b (b+100), while the value of account c has been reduced by 200 units (c − 200). The total amount across all three accounts has, therefore, remained the same.

A comparison between the LOG(b) from Fig. 4.4 and the previously discussed log from Fig. 4.2 shows a major difference: It is now strictly one read (TRX_2: READ) and one write (TRX_2: WRITE) by TRX_2 before TRX_1 gets access to account b and can also read (TRX_1: READ) and write (TRX_1: WRITE) on it. The corresponding precedence graph contains neither READ_WRITE nor WRITE_WRITE edges between the nodes TRX_1 and TRX_2, i.e., it is free of cycles. The two posting transactions, therefore, fulfill the integrity constraint.

In many database applications, the demand for high serializability prohibits the use of entire databases or tables as locking units. Consequently, it is common to define smaller

locking units, such as database excerpts, parts of tables, tuples, or even individual data values. Ideally, *locking units* are defined in a way that allows for *hierarchical dependencies* in lock management. For instance, if a set of tuples is locked by a specific transaction, the superordinate locking units such as the containing table or database must not be completely blocked by other transactions during the lock's validity. When an object is put under an exclusive lock, locking hierarchies can be used to automatically evaluate and mark superordinate objects accordingly.

Various locking modes are also important. The most basic classification of locks is the dichotomy of read-locks and write-locks. Read-locks (or shared locks) grant read-only access for the object to a transaction, while write-locks (or exclusive locks) permit read and write access to the object.

Time stamps that allow for strictly ordered object access according to the age of the transactions are another pessimistic method ensuring serializability. Such time tracking methods allow to keep the chronological order of the individual operations within the transactions and, therefore, avoid conflicts.

4.2.4 Optimistic Methods

Optimistic methods are based on the assumption that conflicts between concurring transactions will be rare occurrences. No locks are set initially in order to increase the degree of synchronization and reduce wait times. Before transactions can conclude successfully, they are validated retroactively.

Transactions with *optimistic concurrency control* have three parts: a *read phase*, a *validation phase*, and a *write phase*. During the read phase, all required objects are read, saved to a separate transaction workspace, and processed there, without any preventative locks being placed. After processing, the validation phase is used to check whether the applied changes conflict with any other transactions. The goal is to check currently active transactions for compatibility and absence of conflicts. If two transactions block each other, the transaction currently in the validation phase is deferred. In the case of successful validation, all changes from the workspace are entered into the database during the write phase.

The use of transaction-specific workspaces increases concurrency in optimistic methods, since reading transactions do not impede each other. Checks are only necessary before writing back changes. This means that the read phases of multiple transactions can run simultaneously without any objects being locked. Instead, the validity of the objects in the workspace, i.e., whether they still match the current state of the database, must be confirmed in the validation phase.

For the sake of simplicity, we will assume that validation phases of different transactions do not overlap. To ensure this, the time the transaction enters the validation phase is marked. This allows for both the start times of validation phases and the transactions themselves to be sorted chronologically. Once a transaction enters the validation phase, it is checked for serializability.

The procedure to do so in optimistic concurrency control is as follows: Let TRX_t be the transaction to be validated and TRX_1 to TRX_k be all concurrent transactions that have already been validated during the read phase of TRX_t. All other transactions can be ignored, since they are handled in strict chronological order. All objects read by TRX_t must be validated, since they could have been modified by any of the critical transactions TRX_1 to TRX_k in the meantime. The set of objects read by TRX_t is labeled READ_SET(TRX_t), and the set of objects written by the critical transactions is labeled WRITE_SET(TRX_1,…,TRX_k). This gives us the following serializability condition:

▶ **Optimistic concurrency control** In order for the transaction TRX_t to be serializable in optimistic concurrency control, the sets READ_ SET(TRX_t) and WRITE_ SET(TRX_1,…,TRX_k) must be *disjoint*.

For a more practical example, we will revisit the two posting transactions TRX_1 and TRX_2 from Fig. 4.1, with the assumption that TRX_2 was validated before TRX_1. To assess whether TRX_1 is serializable in this scenario, we compare the objects read by TRX_1 and those written by TRX_2 (Fig. 4.5) to see that object b is part of both sets, i.e., READ_SET(TRX_1) and WRITE_SET(TRX_2) overlap, thereby violating the serializability condition. The posting transaction TRX_1 has to be rolled back and restarted.

Optimistic methods can be improved by preventatively ensuring the disjointness of the sets READ_SET and WRITE_SET, using the validation phase of a transaction TRX_t to check whether it will modify any objects that have already been read by other transactions. This assessment method limits the validation effort to transactions that actually make changes to database contents.

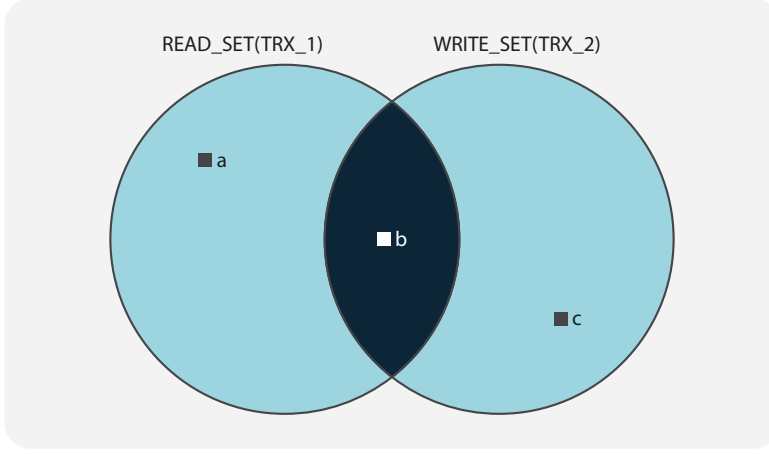


Fig. 4.5 Serializability condition for TRX_1 not met

4.2.5 Troubleshooting

Various errors can occur during database operation and will normally be mitigated or corrected by the database system itself. Some error cases, such as integrity violations or deadlocks have already been mentioned in the sections on concurrency control. Other issues may be caused by operating systems or hardware, for instance when data remains unreadable after a save error on an external medium.

The *restoration of a correct database state after an error* is called recovery. It is essential for recovery to know where an error occurred: in an application, in the database software, or in the hardware. In the case of integrity violations or after an application program "crashes", it is sufficient to roll back and then repeat one or several transactions. With severe errors, it may be necessary to retrieve earlier saves from backup archives and restore the database state by partial transaction reruns.

In order to roll back transactions, the database system requires certain information. Usually, a copy of an object (called *before image*) is written to a *log file*[3] before the object is modified. In addition to the object's old values, the file also contains markers signaling beginning and end of the transaction. In order for the log file to be used efficiently in the case of errors, *checkpoints* are set either based on commands in the application program or for certain system events. A system-wide checkpoint contains a list of the transactions active up until that time. If a restart is needed, the database system merely has to find the latest checkpoint and reset the unfinished transaction, e.g., with the SQL command ROLLBACK.

This procedure is illustrated in Fig. 4.6: After system failure, the log file must be read backwards until the last checkpoint. Of special interest are those transactions that had not been able to indicate their correct conclusion with an EOT (end of transaction) marker, such as the transactions TRX_2 and TRX_5 in our example. For them, the previous database state must be restored with the help of the log file (*undo*). For TRX_5, the file must be read back until the BOT (beginning of transaction) marker in order to retrieve the transaction's before image. Regardless of the type of checkpoint, the newest state (*after image*) must be restored for at least TRX_4 (*redo*).

The recovery of a database after a defect in an external storage medium requires a backup of the database and an inventory of all updates since the creation of the backup copy. Backups are usually made before and after the end-of-day processing, since they are quite time-consuming. During the day, changes are recorded in the log file, with the most up-to-date state for each object being listed.

Securing databases requires a clear-cut *disaster prevention* procedure on the part of the responsible database specialists. Backup copies are usually stored in generations, physically separate, and sometimes redundant. The creation of backup files and the

[3]This log file is not to be confused with the log from Sect. 4.2.2.

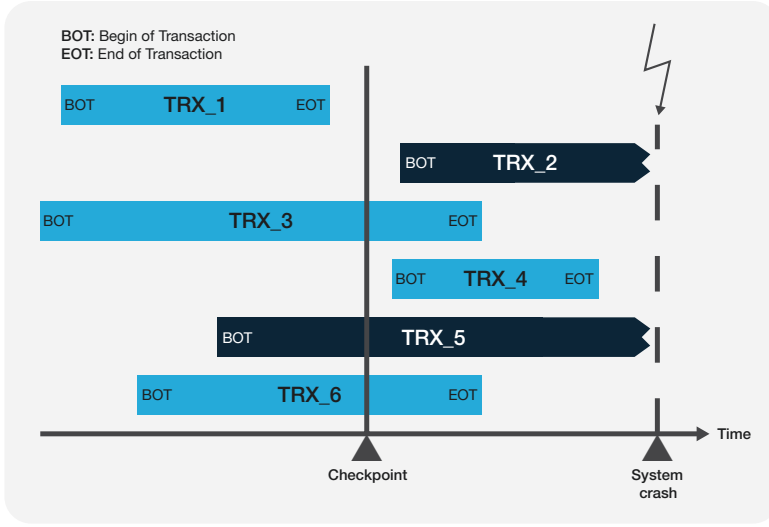


Fig. 4.6 Restart of a database system after an error

removal of old versions must be fully documented. In the case of errors or for disaster drills, the task is to restore current data from backup files and logged changes within a reasonable timeframe.

4.3 Consistency in Massive Distributed Data

4.3.1 BASE and the CAP Theorem

It has become clear in practice that for large and distributed data storage systems, consistency cannot always be the primary goal; sometimes availability and partition tolerance take priority.

In relational database systems, transactions at the highest isolation level are always atomic, consistent, isolated, and durable (ACID, Sect. 4.2.1). Web-based applications, on the other hand, are geared towards high availability and the ability to continue working if a computer node or a network connection fails. Such partition tolerant systems use replicated computer nodes and a softer consistency requirement called BASE (basically available, soft state, eventually consistent). This allows replicated computer nodes to temporarily hold diverging data versions and only be updated with a delay.

During a symposium in 2000, Eric Brewer of the University of California, Berkeley, presented the hypothesis that the three properties of consistency, availability, and partition tolerance cannot exist simultaneously in a massive distributed computer system.

- **Consistency (C)**: When a transaction changes data in a distributed database with replicated nodes, all reading transactions receive the current state, no matter from which node they access the database.
- **Availability (A)**: Running applications operate continuously and have acceptable response times.
- **Partition tolerance (P)**: Failures of individual nodes or connections between nodes in a replicated computer network do not impact the system as a whole, and nodes can be added or removed at any time without having to stop operation.

This hypothesis was later proven by researchers at MIT in Boston and established as the CAP theorem.

▶ **CAP theorem** The CAP theorem states that in any massive distributed data management system, only two of the three properties consistency, availability, and partition tolerance can be ensured.

In short, massive distributed systems can have a combination of either consistency and availability (CA), consistency and partition tolerance (CP), or availability and partition tolerance (AP), but it is impossible to have all three at once (Fig. 4.7).

Use cases of the CAP theorem may include:

- Stock exchange systems requiring consistency and availability (CA), which are achieved by using relational database systems following the ACID principle.
- Country-wide networks of ATMs, which still require consistency, but also partition tolerance, while somewhat long response times are acceptable (CP); distributed and replicated relational or NoSQL systems supporting CP are best suited for this scenario.
- The domain name system (DNS) internet service is used to resolve URLs into numerical IP addresses in TCP/IP (transmission control protocol/internet protocol) communication and must, therefore, always be available and partition tolerant (AP), which is a task that requires NoSQL data management systems, since a relational database system cannot provide global availability and partition tolerance.

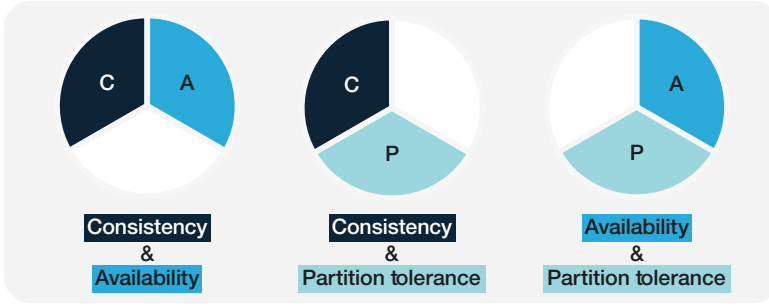


Fig. 4.7 The three possible combinations under the CAP theorem

4.3.2 Nuanced Consistency Settings

Ideally, there would be only one approach to ensuring consistency in a distributed system. Whenever a change is made, all reading transactions see the change and are certain to get the current state. For instance, if a hotel chain offers online reservation via their website, any bookings are immediately recognized by all reading transactions and double bookings are prevented.

However, the CAP theorem has taught us that in networks of replicated computer nodes, only two out of three corresponding properties can be achieved at any time. International hotel chains commonly focus on AP, meaning they require high availability and partition tolerance. In exchange, they accept that bookings are made according to the BASE principle. There are other possible refinements that can be configured based on the following parameters:

- N = number of replicated nodes or number of copies in the cluster
- R = number of copies to be read (successful read)
- W = number of copies to be written (successful write)

With these three parameters N, R, and W, it is possible to formulate four basic options for nuanced consistency control. Figure 4.8 gives an overview of those variants for a

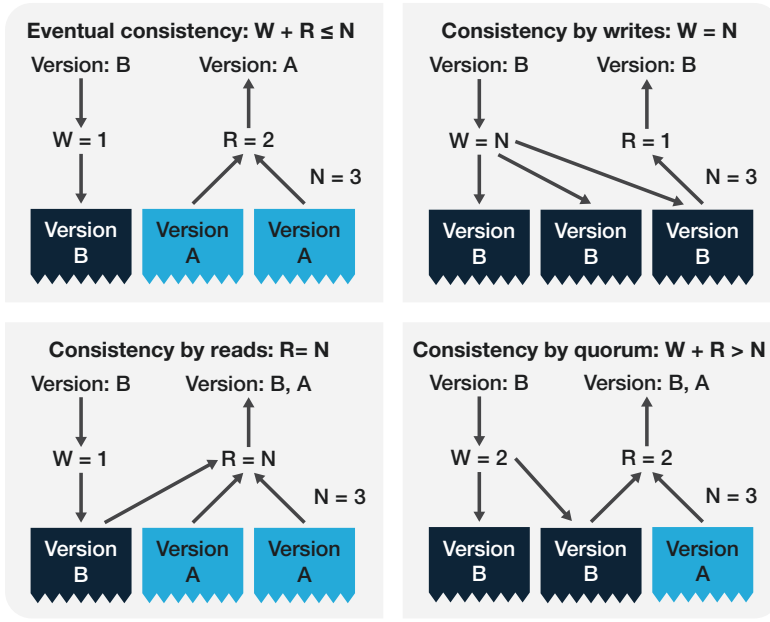


Fig. 4.8 Ensuring consistency in replicated systems

sample case of three replicated nodes (N = 3). Initially, all nodes hold the object version A, before some nodes are subsequently overwritten with the new version B. The issue at hand is how reading programs can identify current versions if writing programs make modifications.

The first option is formulated as $W+R \leq N$. In the example in Fig. 4.8 (top left) the parameters are set to N = 3, W = 1, and R = 2; W = 1 means that at least one node must be written successfully, while R2 requires at least two nodes to be read successfully. In the node accessed by the writing program, the old version A is replaced with the new version B. In the worst case scenario, the reading program accesses the other two nodes and receives the old version A from both of them. This option is, therefore, an example for eventual consistency.

One alternative is "consistency by writes", in which W must match the number of replicated nodes, i.e., W = N (Fig. 4.8, top right). Successful write operations replace version A with the new version B in all three nodes. When a reading program accesses any node, it will always get the current version B.

Option three is called "consistency by reads", since the number of reads equals the number of nodes (Fig. 4.8, bottom left). The new version B is only written on one node, so the consultation of all three nodes by a reading operation returns both the current version B and the old version A. When a transaction receives multiple read results, such as versions A and B in this case, it has to establish the chronological order of the results, i.e., whether it is A before B (A < B) or B before A (B < A), in order to determine which is the newest. This is done with the help of vector clocks (Sect. 4.3.3).

The fourth and final case is "consistency by quorum" with the formula W + R > N (Fig. 4.8, bottom right). In our example, both parameters W and R are set to two, i.e., W = 2 and R = 2. This requires two nodes to be written and two nodes to be read successfully. The read operation once again definitely returns both versions A and B so that the chronological order must be determined using vector clocks.

4.3.3 Vector Clocks for the Serialization of Distributed Events

In distributed systems, various events may occur at different times due to concurring processes. Vector clocks can be used to bring some order into these events. They are not time-keeping tools, but counting algorithms allowing for a partial chronological ordering of events in concurrent processes.

Below, we will look at concurrent processes in a distributed system. A vector clock is a vector with k components or counters C_i with $i = 1 \ldots k$, where k equals the number of processes. Each process P_i, therefore, has a vector clock $V_i = [C_1, \ldots, C_k]$ with k counters.

A vector clock works along the following steps:

- Initially, all vector clocks are set to zero, i.e., $V_i = [0,0,\ldots,0]$ for all processes P_i and counters C_k.

- In each interprocess message, the sender includes its own vector clock for the recipient.
- When a process receives a message, it increments its own counter C_i in its vector by one, i.e., $C_i = C_i + 1$. It also merges its own updated vector V_i with the received vector W component by component by keeping the higher of two corresponding counter values, i.e., $V_i[j] = \max(V_i[j], W[j])$ for all $j = 1 \ldots k$.

Figure 4.9 shows a possible scenario with three concurrent processes P_1, P_2, and P_3.

Process P_3 includes the three events B, D, and E in chronological order. It increments its own counter C_3 in its vector clock by one for each event, resulting in the vector clocks [0,0,1] for event B, [0,0,2] for event D, and [0,0,3] for event E.

In process P_1, event A occurs first and the process' counter C_1 is raised by one in its vector clock V_1, which is then [1,0,0]. Next, P_1 sends a message M_1 to process P_2, including its current vector clock V_1. Event C in process P_2 first updates the process' own vector clock V_2 to [0,1,0] before merging it with the newly received vector clock $V_1 = [1,0,0]$ into [1,1,0].

Similar mergers are executed for the messages M_2 and M_3. First, the processes' vector clocks V_2/V_1 are incremented by one in the process' own counter, then the maximum of the two vector clocks to be merged is determined and included. This results in the vector clocks $V_2 = [1,2,3]$ (since $[1,2,3] = \max([1,2,0], [0,0,3])$) for event F and $V_1 = [3,2,3]$ for event G.

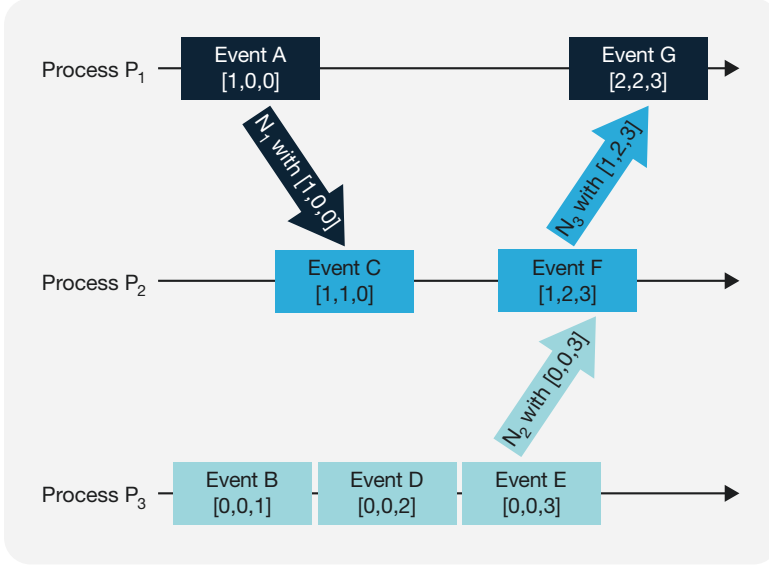


Fig. 4.9 Vector clocks showing causalities

Causality can be established between two events in a distributed system: Event X happened before event Z if the vector clock $V(X) = [X_1, X_2, \ldots, X_k]$ of X is less than the vector clock $V(Y) = [Y_1, Y_2, \ldots, Y_k]$ of Y. In other words:

▶ **Causality principle cased on vector clocks** Event X happened before Event Y (or $X < Y$) if $X_i < Y_i$ for all $i = 1 \ldots k$ and if there is at least one j where $X_j < Y_j$.

In Fig. 4.9, it is clear that event B took place before event D, since the corresponding vector clocks [0,0,1] and [0,0,2] meet the abovementioned condition.

Comparing the events F and G, we can also see from their vector clocks [1,2,3] and [2,2,3] that F happened before G. The first counter of the vector clock V(F) is less than the first counter of V(G), and the other components are identical, and $[1,2,3] < [2,2,3]$ in the vector clocks means a causality $F < G$.

Now assume two fictional vector clocks $V(S) = [3,1,1]$ for an event S and $V(T) = [1,1,2]$ for an event T. These two vector clocks are not comparable, since neither $S < T$ nor $T < S$ is true. The two events are concurrent, and no causality can be established.

Vector clocks are especially suitable for massive distributed and replicated computer structures. Since actual time clocks are hard to synchronize in global networks, vector clocks are used instead, Including as many components as there are replicas.

During the distribution of replicas, vector clocks allow us to determine which version is the newer and, therefore, more current one. For the two options "consistency by reads" and "consistency by quorum" in Sect. 4.3.2, read operations returned both the versions A and B. If these two versions have vector clocks, the causality condition described above can be applied to conclude that $A < B$, i.e., B is the newer version.

4.4 Comparing ACID and BASE

There are some major differences between the ACID (atomicity, consistency, isolation, durability) and BASE (basically available, soft state, eventually consistent) approaches, as summarized in Fig. 4.10.

Relational database systems are strictly based on ACID, meaning that consistency is ensured at any time in both centralized and distributed systems. Distributed relational database systems require a coordinating program that implements all changes to table contents in full and creates a consistent database state. In the case of errors, the coordinating program makes sure that the distributed database is not affected in any way and the transaction can be restarted.

NoSQL systems support ensuring consistency in various ways. Generally, changes in massive distributed data storage systems are written on the source node and replicated to all other nodes. However, this replication may come with a slight delay, so it is possible for nodes to not have the current database state available when accessed by user queries.

ACID	BASE
Consistency is the top priority (strong consistency)	Consistency is ensured only eventually (weak consistency)
Mostly pessimistic concurrency control methods with locking protocols	Mostly optimistic concurrency control methods with nuanced setting options
Availability is ensured for moderate volumes of data	High availability and partition tolerance for massive distributed data storage
Some integrity restraints (e.g., referential integrity) are ensured by the database schema	Some integrity restraints (e.g., referential integrity) are ensured by the database schema

Fig. 4.10 Comparing ACID and BASE

Individual nodes in the computer network are usually accessible (basically available), but may not have been properly updated yet (eventually consistent), i.e., they may be in a soft state.

Relational database systems commonly use pessimistic concurrency control procedures which require locks to be placed and released according to the two-phase locking protocol (Sect. 4.2.3) for the operations of a transaction. If database applications execute disproportionately fewer changes than queries, optimistic methods (Sect. 4.2.4) may be deployed. If conflicts arise, the respective transactions must be restarted.

Massive distributed data management systems focused on availability and partition tolerance can only provide consistent states with a delay according to the CAP theorem. Moreover, placing and removing locks on replicated nodes would take an exorbitant effort. Most NoSQL systems, therefore, use optimistic concurrency control.

In terms of availability, relational database systems are on par with their alternatives up to a certain amount of data and distribution. Big Data applications, however, are generally based on NoSQL systems that offer high availability in addition to either partition tolerance or consistency.

All relational database systems require the explicit specification of tables, attributes, domains, keys, and other integrity constraints, which are then stored in the system catalog. Rules for referential integrity must be defined in the database schema (Sect. 3.8). Queries and changes using SQL rely on that information and cannot be executed without it. Most NoSQL systems do not have an explicit database schema, since changes can happen at any time in the semistructured or unstructured data.

Some NoSQL systems allow for more nuanced settings on how to ensure consistency, resulting in some fuzzy lines between ACID and BASE, as illustrated in Sect. 4.3.2.

4.5 Further Reading

Gray and Reuter (1993), Weikum (1988), and Weikum and Vossen (2002) provide detailed explanations of transaction concepts. The term ACID was first introduced by Härder and Reuter (1983). 2PL was defined by Eswaran et al. (1976) at the IBM Research Lab in San Jose. Bernstein et al. (1987) describe multi-user operation and recovery procedures, while the dissertation of Schaarschmidt (2001) presents concepts and languages for database backups. Another dissertation, this one by Störl (2001), is concerned with backup and recovery in database systems. Reuter (1981) explains troubleshooting procedures for database systems, and both Castano et al. (1994) and Basta and Zgola (2011) discuss various methods for securing databases.

In 2000, Eric Brewer held a keynote presentation at the Symposium on Principles of Distributed Computing, which is credited as the first introduction of the CAP theorem (Brewer 2000). The theorem was proven 2 years later by Seth Gilbert and Nancy Lynch at MIT (Gilbert and Lynch 2002). Werner Vogels of Amazon.com described the many facets of consistency in the CACM (Communications of the Association for Computing Machinery) journal and coined the term "eventually consistent" (Vogels 2009). The nuanced consistency settings from Sect. 4.3.2 are based on the key-value store Riak and were compiled from the NoSQL database book by Redmond and Wilson (2012) and the manufacturer's information on Riak (2014). The approach of using quorums in distributed systems is based on the work of Gifford (1979), among others.

References

Basta A., Zgola M.: Database Security. Cengage Learning, Australia (2011)

Bernstein P. A., Hadzilacos V., Goodman N.: Concurrency Control and Recovery in Database Systems. Addison Wesley, Reading (1987)

Brewer E.: Keynote—Towards robust distributed systems. 19th ACM Symposium on Principles of Distributed Computing, Portland, Oregon, 16–19 July 2000

Castano S., Fugini M.G., Martella G., Samarati P.: Database Security. Addison Wesley, Wokingham (1994)

Eswaran, K.P., Gray, J., Lorie, R.A., Traiger, I.L.: The notion of consistency and predicate locks in a data base system. Commun. ACM **19**(11), 624–633 (1976)

Gifford D.K.: Weighted voting for replicated data. Proc. of the seventh ACM Symposium on Operating Systems Principles (SOSP'79), Pacific Grove, CA, 10–12 December 1979, pp. 150–162

Gilbert, S., Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. Massachusetts Institute of Technology, Cambridge (2002)

Gray J., Reuter A.: Transaction Processing—Concepts and Techniques. Morgan Kaufmann, San Mateo (1993)

Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15**(4), 287–317 (1983)

Redmond E., Wilson J.R.: Seven Databases in Seven Weeks—A Guide to Modern Databases and the NoSQL Movement. The Pragmatic Bookshelf, Dallas (2012)
Reuter A: Fehlerbehandlung in Datenbanksystemen. Hanser, München (1981)
Riak: Open source distributed database. http://basho.com/riak/ (2014). Accessed 18 Dec 2014
Schaarschmidt R.: Archivierung in Datenbanksystemen—Konzepte und Sprache. Teubner, Stuttgart (2001)
Störl, U.: Backup und Recovery in Datenbanksystemen—Verfahren, Klassifikation, Implementierung und Bewertung. Teubner, Stuttgart (2001)
Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009)
Weikum G.: Transaktionen in Datenbanksystemen—Fehlertolerante Steuerung paralleler Abläufe. Addison Wesley, Bonn (1988)
Weikum G., Vossen G.: Transactional Information Systems—Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann, San Francisco (2002)

System Architecture

5

5.1 Processing of Homogeneous and Heterogeneous Data

Throughout the 1950s and 1960s, file systems were kept on secondary-storage media (tapes, drum memories, magnetic disks), before database systems became available on the market in the 1970s. Those file systems allowed for random, or direct, access to the external storage, i.e., specific records could be selected by using an address, without the entirety of records needing to be checked first. The access address was determined via an index or a hash function (Sect. 5.2.2).

The mainframe computers running these file systems were largely used for technical and scientific applications (computing numbers). With the emergence of database systems, computers also took over in business contexts (computing numbers and text) and became the backbone of administrative and commercial applications, since database systems support consistency in multi-user operation (ACID, Sect. 4.2.1). Today, many information systems are still based on the relational database technology that replaced most of the previously used hierarchic or network-like database systems.

Relational database systems use only tables to store and handle data. A table is a set of records that can flexibly process structured data.

Structured data strictly adheres to a well-defined data structure with a focus on the following properties:

- **Schema**: The structure of the data must be communicated to the database system by specifying a schema (see the SQL command CREATE TABLE in Chap. 3). In addition to table formalization, integrity constraints are also stored in the schema (cf., e.g., the definition of referential integrity and the establishment of appropriate processing rules, Sect. 3.7).

A. Meier and M. Kaufmann, *SQL & NoSQL Databases,*
https://doi.org/10.1007/978-3-658-24549-8_5

- **Data types**: The relational database schema guarantees that for each use of the database, the data manifestations always have the set data types (e.g., CHARACTER, INTEGER, DATE, TIMESTAMP, etc.; see also the SQL tutorial at www.sql–nosql.org). To do so, the database system consults the system tables (schema information) at every SQL invocation. Special focus is on authorization and data protection rules, which are checked via the system catalog (see VIEW concept and privilege assignment via GRANT and REVOKE in Sect. 3.8, as well as the SQL tutorial at www.sql-nosql.org).

Relational databases, therefore, mostly process structured and formatted data. In order to meet specific requirements in the fields of office automation, technology, and web applications (among others), SQL has been extended by data types and functions for alphabetical strings (CHARACTER VARYING), bit sequences (BIT VARYING, BINARY LARGE OBJECT), and text fragments (CHARACTER LARGE OBJECT) (see the SQL tutorial at www.sql-nosql.org). The integration of XML (eXtensible Markup Language) is also supported. These additions resulted in the definition of semistructured and unstructured data.

Semistructured data is defined as follows:

- They consist of a set of data objects whose structure and content is subject to continuous changes.
- Data objects are either atomic or composed of other data objects (complex objects).
- Atomic data objects contain data values of a specified data type.

Data management systems for semistructured data work without a fixed database schema, since structure and content change constantly. A possible use case is content management systems for websites which can flexibly store and process webpages and multimedia objects. Such systems require extended relational database technology (Chap. 6), XML databases, or NoSQL databases (Chap. 7).

A *data stream* is a continuous flow of digital data with a variable data rate (records per unit of time). Data within a data stream is sorted chronologically and often given a timestamp. Besides audio and video data streams, this can also be a series of measurements which are analyzed with the help of analysis languages or specific algorithms (language analysis, text analysis, pattern recognition, etc.). Unlike structured and semistructured data, data streams can only be analyzed sequentially.

Figure 5.1 shows a simple use case for data streams. The setting is a multi-item English auction via an electronic bidding platform. At this auction, bidding starts at a set minimum. Participants can make multiple bids that have to be higher than the previous highest bid. Since electronic auctions have no physical location, time and duration of the auction are set in advance. The bidder who makes the highest bid during the set time wins the auction.

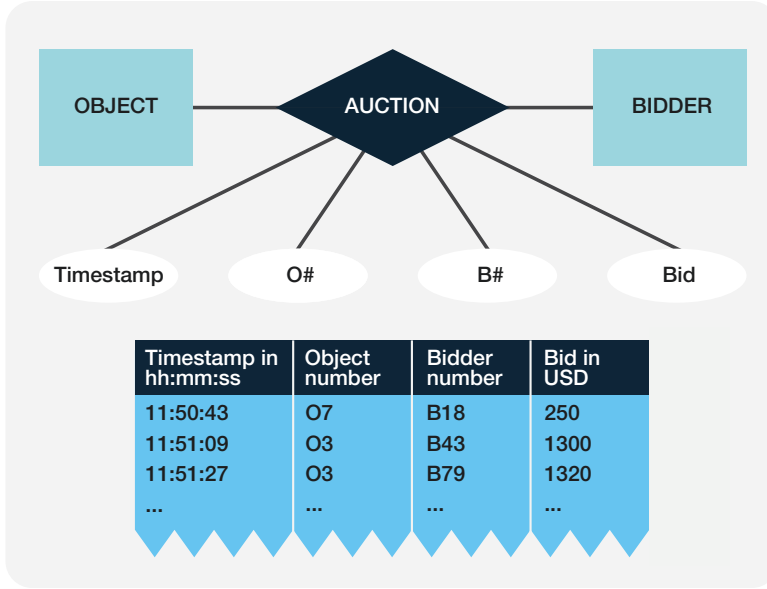


Fig. 5.1 Processing a data stream

Any AUCTION can be seen as a relationship set between the two entity sets OBJECT and BIDDER. The corresponding foreign keys O# and B# are complemented by a timestamp and the sum offered (e.g., in USD) per bid. The data stream is used to show bidders the current standing bids during the auction. After the auction is over, the highest bids are made public, and the winners of the individual items are notified. The data stream can then be used for additional purposes, for instance bidding behavior analyses or disclosure in the case of legal contestation.

Unstructured data are digital data without any fixed structure. This includes multimedia data such as continuous text, music files, satellite imagery, or audio/video recordings. Unstructured data are often transmitted to a computer via digital sensors, for example in the data streams explained above, which can sequentially transport structured and/or unstructured data.

The processing of unstructured data or data streams calls for specially adapted software packages. NoSQL databases or specific data stream management systems are used to fulfill the requirements of Big Data processing.

The next sections discuss several architectural aspects of SQL and NoSQL databases. Section 5.2 focuses on relational database technology. It shows how set-oriented queries can be processed and optimized in SQL. Section 5.3 is about an important parallelization method called MapReduce and explains how queries on massive distributed NoSQL databases can be executed efficiently. In Sect. 5.4, we look at storage and access structures that are used in both SQL and NoSQL databases, albeit in different forms.

The three kinds discussed there are tree structures, address determination (hashing/consistent hashing), and multidimensional data structures. Section 5.5 provides an example of a layered architecture with well-defined software layers, and Sect. 5.6 discusses why many web-based application systems use relational and nonrelational data storage simultaneously. Sources and literature for further reading are listed in Sect. 5.7.

5.2 Storage and Access Structures

Storage and access structures for relational and nonrelational database systems should be designed to manage data in secondary storage as efficiently as possible. For large amounts of data, the structures used in main storage cannot simply be reproduced on the background memory. Instead, it is necessary to optimize the storage and access structures in order to enable reading and writing contents *on external storage media with as few accesses as possible*.

5.2.1 Indexes and Tree Structures

An *index* of an attribute is an access structure that efficiently provides, in a specific order for each attribute value, the internal addresses of all records containing that attribute value. It is similar to the index of a book, where each entry—listed in alphabetical order—is followed by the numbers of the pages containing it.

For an example, we shall look at an index of the Name attribute for the EMPLOYEE table. This index, which remains invisible to standard users, can be constructed with the following SQL command:

```
CREATE INDEX IX1 ON EMPLOYEE(NAME);
```

For each name in the EMPLOYEE table, sorted alphabetically, either the identification key E# or the internal address of the employee tuple is recorded. The database system uses this index of employee names for corresponding queries or when executing a join. In this case, the Name attribute is the *access key*.

Tree structures can be used to store records or access keys and to *index* attributes in order to increase efficiency. For large amounts of data, the root, internal, and leaf nodes of the tree are not assigned individual keys and records, but rather entire *data pages*. In order to find a specific record, the tree then has to be searched.

With central memory management, the database system usually uses binary trees in the background in which *the root node and each internal node has two subtrees*. Such trees cannot be used unlimitedly for storing access keys or records for extensive databases, since their height grows exponentially for larger amounts of data; large trees, however, are impractical for searching and reading data content on external storage media, since they require too many page accesses.

The *height of a tree*, i.e., the distance between the root node and the leaves, is an *indicator for the number of accesses* required on external storage media. To keep the number of external accesses as low as possible, it is common to make the storage tree structures for database systems grow in width instead of height. One of the most important of these tree structures is the *B-tree* (Fig. 5.2).

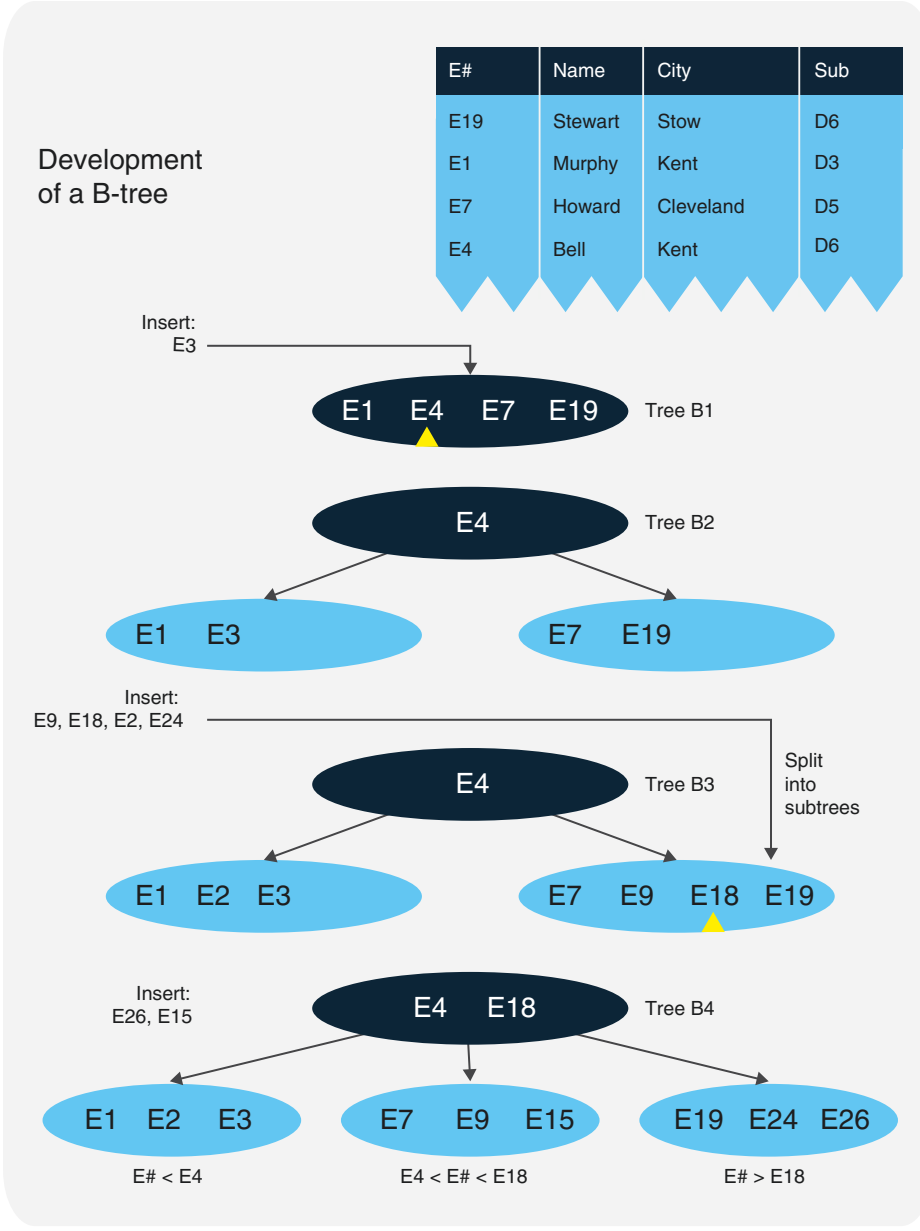


Fig. 5.2 B-tree with dynamic changes

A B-tree is a tree whose *root node and internal nodes generally have more than two subtrees*. The data pages represented by the individual internal and leaf nodes should not be empty, but ideally filled with key values or entire records. They are, therefore, usually required to be filled at least halfway with records or keys (except for the page associated with the root node).

▶ **B-tree** A tree is a *B-tree of the n-th order* if

- it is fully balanced (the paths from the root to each leaf have the same length) and
- each node (except for the root node) has at least n and at the most 2*n entries in its data page.

That second condition also means that, since every node except the root node has at least n entries, each node has at least n subtrees. On the other hand, each node has a maximum of 2*n entries, i.e., no node of a B-tree can have more than 2*n subtrees.

Assume, for instance, that the key E# from the EMPLOYEE table is to be stored in a B-tree of the order n = 2 as an access structure, which results in the tree shown in Fig. 5.2.

Nodes and leaves of the tree cannot contain more than four entries due to the order 2. Apart from the keys, we will assume that the pages for the nodes and leaves hold not only key values, but also pointers to the data pages containing the actual records. This means that the tree in Fig. 5.2 represents an access tree, not the data management for the records in the EMPLOYEE table.

In our example, the root node of the B-tree contains the four keys E1, E4, E7, and E19 in numerical order. When the new key E3 is added, the root node must be split because it cannot hold any more entries. The split is done in a way that produces a balanced tree. The key E4 is declared the new root node, since it is in between two equal halves of the remaining key set. The left subtree is formed of key values that meet the condition "E# lower than E4" (in this case, E1 and E3), the right subtree consists of key values where "E# higher than E4" (i.e., E7 and E19). Additional keys can be inserted in the same way, while the tree retains a fixed height.

The database system searches for individual keys top-down, e.g., if the candidate key E15 is requested from the B-tree B4 in Fig. 5.2, it checks against the entries in the root node. Since E15 lies between the keys E4 and E18, it selects the corresponding subtree (in this case, only one leaf node) and continues the search until it finds the entry in the leaf node. In this simple example, the search for E15 requires only two page accesses, one for the root node and one for the leaf node.

The height of a B-tree determines the access times for keys as well as the data associated with a (search) key. The access times can be reduced by increasing the branching factor of the B-tree.

Another option is a *leaf-oriented B-tree* (commonly called B + tree), where the actual records are never stored in internal nodes but only in leaf nodes. The internal roots contain only key entries in order to keep the tree as low as possible.

5.2.2 Hashing Methods

Key hashing or simply hashing is an address determination procedure that is at the core of any distributed data and access structures. *Hash functions* map a set of keys on a set of addresses forming a contiguous address space.

A simple hash function assigns a number between 1 and n to each key of a record as its address. This address is interpreted as a relative page number, with each page holding a set number of key values with or without their respective records.

Hash functions must meet the following requirements:

- It must be possible to follow the transformation rule with simple calculations and few resources.
- The assigned addresses must be distributed evenly across the address space.
- The probability of assignment collisions, i.e., the use of identical addresses for multiple keys, must be the same for all key values.

There is a wide variety of hash functions, each of which has its pros and cons. One of the simplest and best-known algorithms is the division method.

▶ **The division method of hashing** Each key is interpreted as an integer by using bit representation. The *hash function H for a key k and a prime number p* is given by the formula

H(k): = k mod p.

The integer "k mod p"—the remainder from the division of the key value k by the prime number p—is used as a relative address or page number. In the division method, the choice of the prime number p determines the memory use and the uniformity of distribution.

Figure 5.3 shows the EMPLOYEE table and how it can be mapped to different pages with the division method of hashing.

In this example, each page can hold four key values. The prime number chosen for p is 5. Each key value is now divided by 5, with the remaining integer determining the page number.

Inserting the key E14 causes problems, since the corresponding page is already full. The key E14 is placed in an *overflow area*. A link from page 4 to the overflow area maintains the affiliation of the key with the co-set on page 4.

There are multiple methods for handling overflows. Instead of an overflow area, additional hash functions can be applied to the extra keys. Quickly growing key ranges or complex delete operations often cause difficulties in overflow handling. In order to mitigate these issues, dynamic hashing methods have been developed.

Such *dynamic hash functions* are designed to keep memory use independent from the growth of keys. Overflow areas or comprehensive redistribution of addresses are mostly rendered unnecessary. The existing address space for a dynamic hash function can be extended either by a specific choice of hashing algorithm or by the use of a page

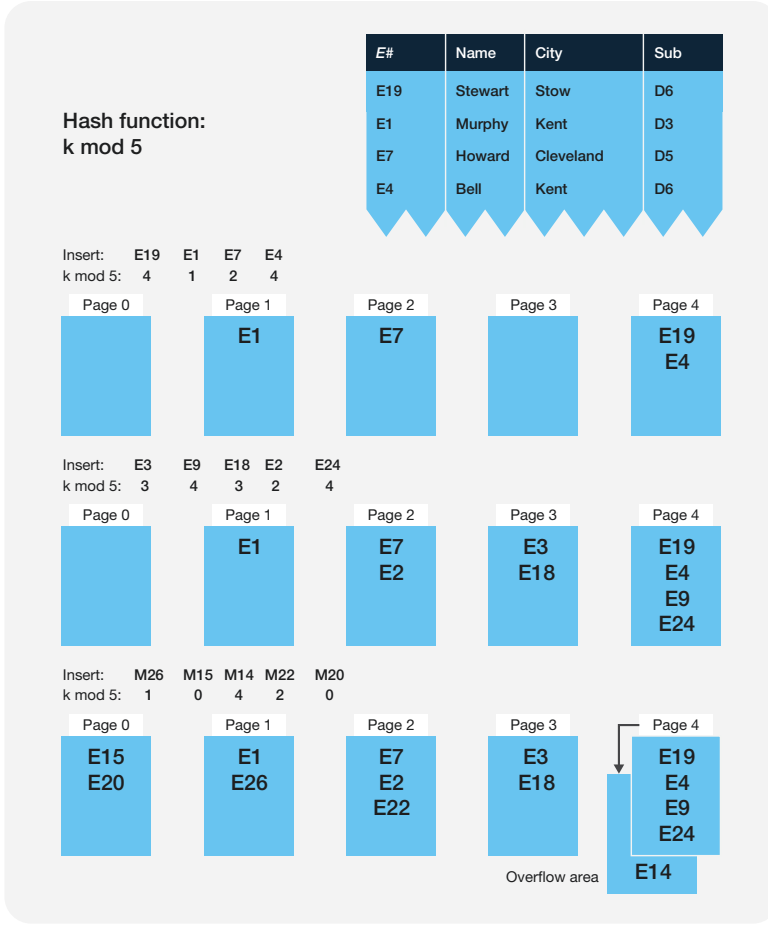


Fig. 5.3 Hash function using the division method

assignment table kept in the main memory, without the need to reload all keys or records already stored.

5.2.3 Consistent Hashing

Consistent hashing functions belong to the family of distributed address calculations (see hashing methods in the previous section). A storage address or hash value is calculated from a set of keys in order to store the corresponding record.

In Big Data applications, the key-value pairs are assigned to different nodes in the computer network. Based on the keys (e.g., term or day), their values (e.g., frequencies) are stored in the corresponding node. The important part is that with consistent hashing, address calculation is used for both the node addresses and the storage addresses of the objects (key-value).

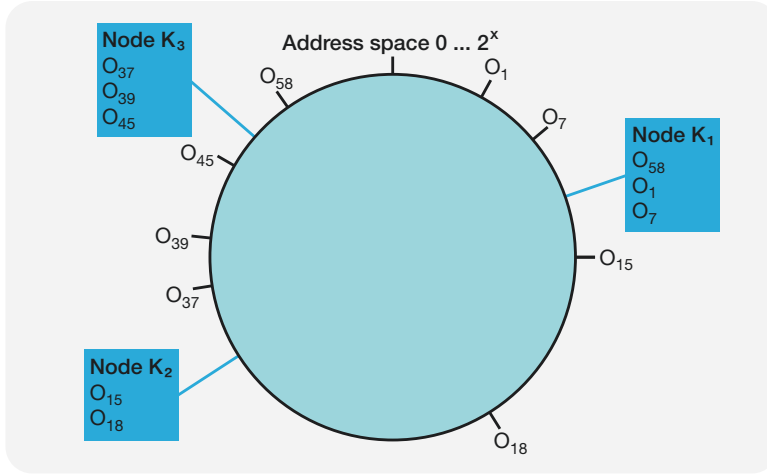


Fig. 5.4 Ring with objects assigned to nodes

Figure 5.4 provides a schematic representation of consistent hashing. The address space of 0 to 2^x key values is arranged in a circle; then a hash function is selected to run the following calculations:

- **Calculation of node addresses**: The nodes' network addresses are mapped to storage addresses using the selected hash function, then entered on the ring.
- **Calculation of object addresses**: The keys of the key-value pairs are transformed into addresses with the hashing algorithm, and the objects are entered on the ring.

The key-value pairs are stored on their respective storage nodes according to a simple assignment rule: The objects are assigned to the next node (clockwise) and managed there.

Figure 5.4 shows an address space with three nodes and eight objects (key-value pairs). The positioning of the nodes and objects results from the calculated addresses. According to the assignment rule, objects O_{58}, O_1, and O_7 are stored on node K_1; objects O_{15} and O_{18} on node K_2; and the remaining three objects on node K_3.

The strengths of consistent hashing best come out in flexible computer structures, where nodes may be added or removed at any time. Such changes only affect objects directly next to the respective nodes on the ring, making it unnecessary to recalculate and reassign the addresses for a large number of key-value pairs with each change in the computer network.

Figure 5.5 illustrates two changes: Node K_2 is removed and a new node K_4 is added. After the local adjustments, object O_{18}, which was originally stored in node K_2, is now stored in node K_3. The remaining object O_{15} is transferred to the newly added node K_4 according to the assignment rule.

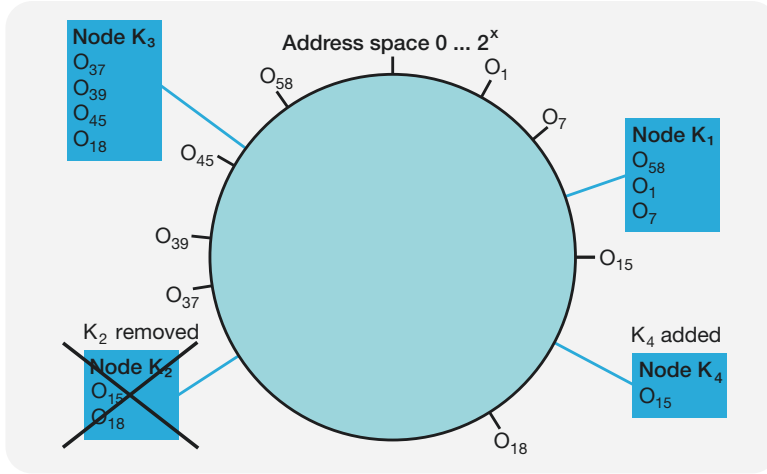


Fig. 5.5 Dynamic changes in the computer network

Consistent hashing can also be used for replicated computer networks. The desired copies of the objects are simply given a version number and entered on the ring. This increases partition tolerance and the availability of the overall system.

Another option is the introduction of virtual nodes in order to spread the objects across nodes more evenly. In this method, the nodes' network addresses are also assigned version numbers in order to be represented on the ring.

Consistent hashing functions are used in many NoSQL systems, especially in implementations of key-value store systems.

5.2.4 Multidimensional Data Structures

Multidimensional data structures support access to records with multiple access key values. The combination of all those access keys is called a *multidimensional key*. A multidimensional key is always unique, but does not have to be minimal.

A data structure that supports such multidimensional keys is called a *multidimensional data structure*. For instance, an EMPLOYEE table with the two key parts Employee Number and Year of Birth can be seen as a two-dimensional data structure. The employee number forms one part of the two-dimensional key, but remains unique in itself. The Year attribute is the second part and serves as an additional access key, without having to be unique.

Unlike tree structures, multidimensional data structures are designed so that no one key part controls the storage order of the physical records. A multidimensional data structure is called *symmetrical* if it permits access with multiple access keys without favoring a certain key or key combination. For the sample EMPLOYEE table, both key parts, Employee Number and Year of Birth, should be equally efficient in supporting access for a specific query.

One of the most important multidimensional data structures is the grid file, or bucket grid.

▶ **Grid file** A grid file is a multidimensional data structure with the following properties:

- It supports access with a *multidimensional access key* symmetrically, i.e., no key dimension is dominant.
- It enables reading any record with only *two page accesses*, one on the grid index, the second on the data page itself.

A grid file consists of a grid index and a file containing the data pages. The grid index is a multidimensional space with each dimension representing a part of the multidimensional access key. When records are inserted, the index is partitioned into cells, alternating between the dimensions. Accordingly, the example in Fig. 5.6 alternates between

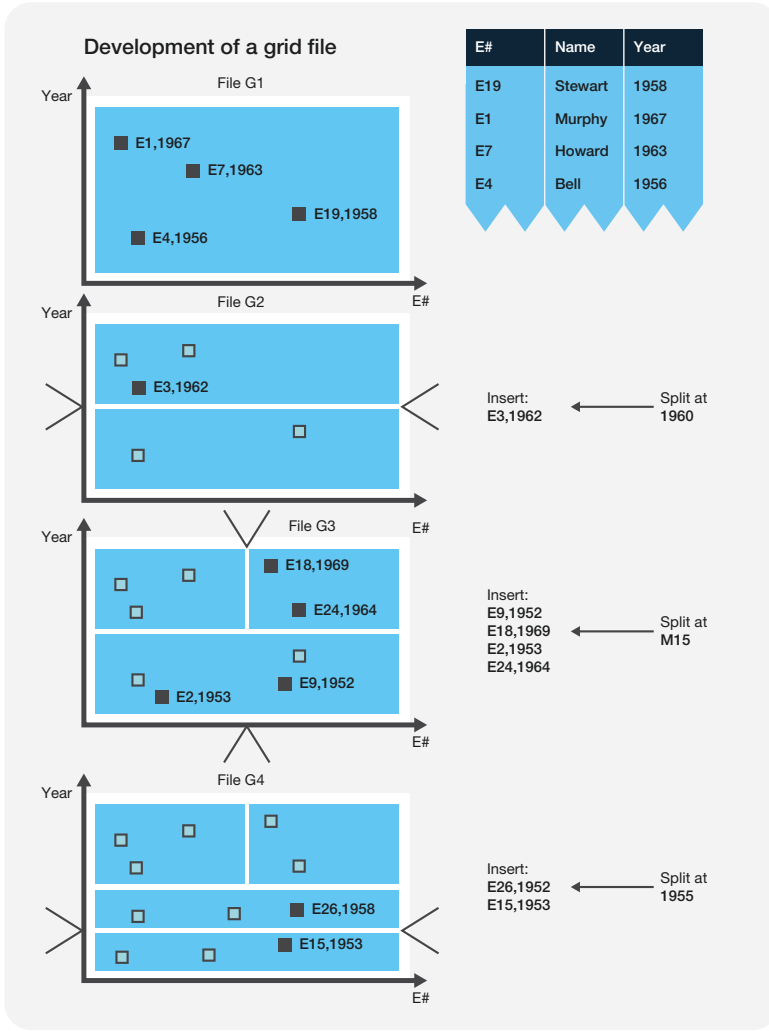


Fig. 5.6 Dynamic partitioning of a grid index

Employee Number and Year of Birth for the two-dimensional access key. The resulting division limits are called the scales of the grid index.

One *cell of the grid index* corresponds to one data page and contains at least n and at the most 2*n entries, n being the number of dimensions of the grid file. Empty cells must be combined with other cells so that the associated data pages can have the minimum number of entries. In our example, data pages can hold no more than four entries (n = 2).

Since the grid index is generally large, it has to be stored in secondary memory along with the records. The set of scales, however, is small and can be held in the main memory. The procedure for accessing a specific record is, therefore, as follows: The system searches the scales with the k key values of the k-dimensional grid file and determines the interval in which each individual part of the search key is located. These intervals describe a cell of the grid index, which can then be accessed directly. Each index cell contains the number of the data page with the associated records, so that one more access, to the data page of the previously identified cell, is sufficient to find whether or not there is a record matching the search key.

The two-disk-access maximum, i.e., no more than two accesses to secondary memory, is guaranteed for the search for any record. The first access is to the appropriate cell of the grid index, the second to the associated data page. As an example, the employee with number E18, born in 1969, is searched in the grid file G4 from Fig. 5.6: The employee number E18 is located in the scale interval E15 to E30, i.e., in the right half of the grid file. The year 1969 can be found between the scales 1960 and 1970, or in the top half. With those scales, the database system finds the address of the data page in the grid index with its first access. The second access, to the respective data page, leads to the requested records with the access keys (E18, 1969) and (E24, 1964).

A k-dimensional grid file supports queries for individual records or record areas. *Point queries* can be used to find a specific record with k access keys. It is also possible to formulate partial queries specifying only a part of the key. With a *range query*, on the other hand, users can examine a range for each of the k key parts. All records whose key parts are in the defined range are returned. Again, it is possible to only specify and analyze a range for part of the keys (partial range query).

The search for the record (E18, 1969) described above is a typical example of a point query. If only the employee's year of birth is known, the key part 1969 is specified for a partial point query. A search for all employees born between 1960 and 1969, for instance, would be a (partial) range query. In the example in Fig. 5.6, this query targets the upper half of grid index G4, so only those two data pages have to be searched. This indexation method allows for the results of range and partial range queries in grid files to be found without the need to sift through the entire file.

In recent years, various multidimensional data structures efficiently supporting multiple access keys symmetrically have been researched and described. The market range of multidimensional data structures for SQL and NoSQL databases is still very limited, but web-based searches are increasing the demand for such storage structures. Especially geographic information systems must be able to handle both topological and geometrical queries efficiently.

5.3 Translation and Optimization of Relational Queries

5.3.1 Creation of Query Trees

The user interfaces of relational database systems are set-oriented, since entire tables or views are provided for the users. When a relational query and data manipulation language is used, the database system has to translate and optimize the respective commands. It is vital that neither the calculation nor the optimization of the query tree require user actions.

▶ **Query tree** Query trees graphically visualize relational queries with the *equivalent expressions of relational algebra*. The leaves of a query tree are the tables used in the query; root and internal nodes contain the algebraic operators.

Figure 5.7 illustrates a query tree using SQL and the previously introduced EMPLOYEE and DEPARTMENT tables. Those tables are queried for a list of cities where the IT department members live:

```
SELECT City
FROM EMPLOYEE, DEPARTMENT
WHERE Sub=D# AND Department_Name='IT'
```

This query can also be expressed algebraically by a series of operators:

$$\text{TABLE} := \pi_{\text{City}} (\sigma_{\text{Department\_Name=IT}} (\text{EMPLOYEE} |X|_{\text{Sub=D\#}} \text{DEPARTMENT}))$$

This expression first calculates a join of the EMPLOYEE and the DEPARTMENT tables via the shared department number. Next, those employees working in the department with the name IT are selected for an intermediate result, and finally, the requested cities are returned with the help of a projection. Figure 5.7 shows this expression of algebraic operators represented in the corresponding query tree.

This query tree can be interpreted as follows: The leaf nodes are the two tables EMPLOYEE and DEPARTMENT used in the query. They are first combined in one internal node (join operator), then reduced to those entries with the department name IT in a second internal node (select operator). The root node represents the projection generating the results table with the requested cities.

Root and internal nodes of query trees refer to either one or two subtrees. If the operator forming a node works with one table, it is called a unary operator; if it affects two tables, it is a binary operator. *Unary operators*, which can only manipulate one table, are the project and select operators (Figs. 3.2 and 3.3). *Binary operators* involving two tables as operands are the set union, set intersection, set difference, Cartesian product, join, and divide operators.

Creating a query tree is the first step in translating and executing a relational database query. The tables and attributes specified by the user must be available in the system

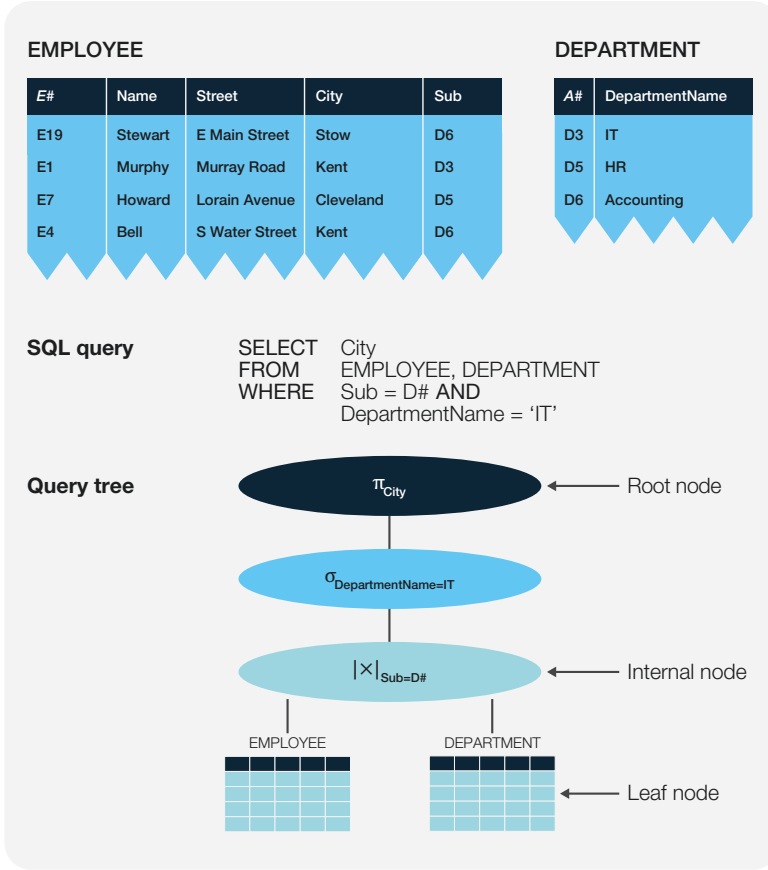


Fig. 5.7 Query tree of a qualified query on two tables

tables before any further processing takes place. The query tree is, therefore, used to check both the query syntax and the user's access permissions. Additional security measures, such as value-dependent data protection, can only be assessed during the runtime.

The second step after this access and integrity control is selection and optimization of access paths; the actual code generation or interpretative execution of the query is step three. With code generation, an access module is stored in a module library for later use; alternatively, an interpreter can take over dynamic control to execute the command.

5.3.2 Optimization by Algebraic Transformation

As demonstrated in Chap. 3, the individual operators of relational algebra can also be combined. If such combined expressions generate the same result despite a different order of operators, they are called *equivalent expressions*. Equivalent expressions allow

for database queries to be optimized with algebraic transformations without affecting the result. By thus reducing the computational expense, they form an important part of the optimization component of a relational database system.

The huge impact of the sequence of operators on the computational expense can be illustrated with the example query from the previous section: The expression

TABLE:= π_{City}
($\sigma_{\text{Department\_Name}=\text{IT}}$
(EMPLOYEE |X| $_{\text{Sub=D\#}}$ DEPARTMENT))
can be substituted with the following equivalent expression, as shown in Fig. 5.8:
TABLE:= π_{City}
($\pi_{\text{Sub,City}}$ (EMPLOYEE)
|x| $_{\text{Sub=D\#}}$
$\pi_{\text{D\#}}$ ($\sigma_{\text{Department\_Name=IT}}$ (DEPARTMENT)))

Here, the first step is the selection ($\sigma_{\text{Department\_Name}=\text{IT}}$) on the DEPARTMENT table, since only the IT department is relevant to the query. Next are two projection operations: one ($\pi_{\text{Sub,City}}$) on the EMPLOYEE table, another ($\pi_{\text{D\#}}$) on the intermediate table with the IT department from step one. Only now is the join operation ($|x|_{\text{Sub=D\#}}$) via the department number executed, before the final projection (π_{City}) on the cities is done. While the end result is the same, the computational expense is significantly lower this way.

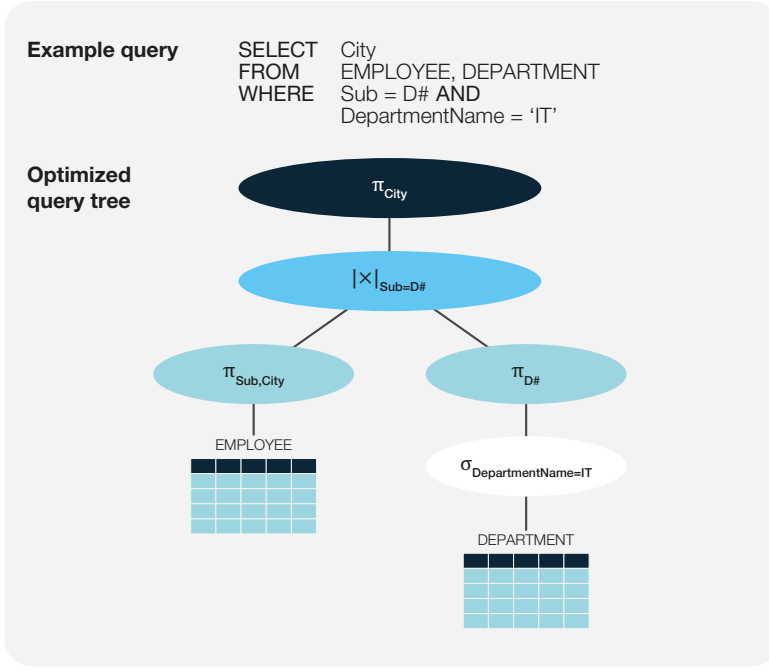


Fig. 5.8 Algebraically optimized query tree

It is generally advisable to position *projection and selection operators in the query tree as close to the leaves as possible* to obtain only small intermediate results before calculating the time-intensive and, therefore, expensive join operators. A successful transformation of a query tree with such a strategy is called *algebraic optimization;* the following principles apply:

- Multiple selections on one table can be merged into one so the selection predicate only has to be validated once.
- Selections should be made as early as possible to keep intermediate results small. To this end, the selection operators should be placed as close to the leaves (i.e., the source tables) as possible.
- Projections should also be run as early as possible, but never before selections. Projection operations reduce the number of columns and often also the tuples.
- Join operators should be calculated near the root node of the query tree, since they require a great deal of computational expense.

In addition to algebraic optimization, the use of efficient storage and access structures (Sect. 5.2) can also achieve significant gains in processing relational queries. For instance, database systems will improve selection and join operators based on the size of the affected tables, sorting orders, index structures, etc. At the same time, an effective model for estimating access costs is vital to decide between multiple possible processing sequences.

Cost formulas are necessary to calculate the computational expense of different database queries, such as sequential searches within a table, searches via index structures, the sorting of tables or subtables, the use of index structures regarding join attributes, or computations of equi-joins across multiple tables. Those cost formulas involve the number of accesses to *physical pages* and creates a weighted gauge for input and output operations as well as CPU (central processing unit) usage. Depending on the computer configuration, the formula may be heavily influenced by access times for external storage media, caches, and main memories, as well as the internal processing power.

5.3.3 Calculation of Join Operators

A relational database system must provide various algorithms that can execute the operations of relational algebra and relational calculus. The selection of tuples from multiple tables is significantly more expensive than a selection from one table. The following section will, therefore, discuss the different join strategies, even though casual users will hardly be able to influence the calculation options.

Implementing a join operation on two tables aims to compare each tuple of one table with all tuples of the other table concerning the join predicate and, when there is a match, insert the two tuples into the results table as a combined tuple. Regarding the

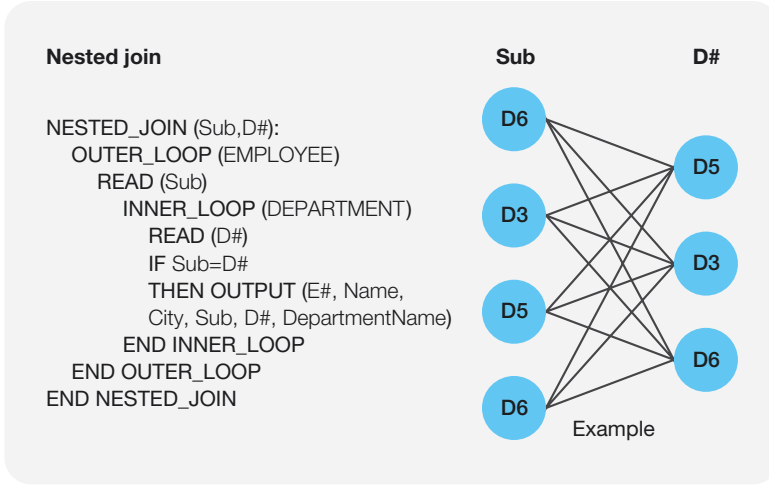


Fig. 5.9 Computing a join with nesting

calculation of equi-joins, there are two basic join strategies: nested join and sort-merge join.

▶ **Nested join** For a nested join between a table R with an attribute A and a table S with an attribute B, *each tuple in R is compared to each tuple in S* to check whether the join predicate R.A = S.B is fulfilled. If R has n tuples and S has m tuples, this requires n times m comparisons.

The algorithm for a nested join calculates the Cartesian product and simultaneously checks whether the join predicate is met. Since we compare all tuples of R in an outer loop with all tuples of S from an inner loop, the expense is quadratic. It can be reduced if an index (Sect. 5.2.1 exists for attribute A and/or attribute B.

Figure 5.9 illustrates a heavily simplified algorithm for a nested join of employee and department information from the established example tables. OUTER_LOOP and INNER_LOOP are clearly visible and show how the algorithm compares all tuples of the EMPLOYEE table to all tuples of the DEPARTMENT table.

For the join operation in Fig. 5.9, there is an index for the D# attribute, since it is the primary key[1] of the DEPARTMENT table. The database system uses the index structure for the department number by not going through the entire DEPARTMENT table tuple by tuple for each iteration of the inner loop, but rather accessing tuples

[1]The database system automatically generates index structures for each primary key; advanced index structures are used for concatenated keys.

directly via the index. Ideally, there is also an index for the Sub (subordinate) attribute of the EMPLOYEE table for the database system to use for optimization. This example illustrates the importance of the selection of suitable index structures by database administrators.

A more efficient algorithm than a nested join is available if the tuples of tables R and S are already sorted physically into ascending or descending order by the attributes A and B of the join predicate, respectively. This may require an internal sort before the actual join operation in order to bring both of the tables into matching order. The computation of the join then merely requires going through the tables for ascending or descending attribute values of the join predicate and simultaneously comparing the values of A and B. This strategy is characterized as follows:

▶ **Sort-merge join** A sort-merge join requires the tables R and S with the join predicate R.A = S.B to be sorted by the attribute values for A of R and B of S, respectively. The algorithm computes the join by making *comparisons in the sorting order*. If the attributes A and B are uniquely defined (e.g., as primary and foreign key), the computational expense is linear.

Figure 5.10 shows a basic algorithm for a sort-merge join. First, both tables are sorted by the attributes used in the join predicate; then the algorithm goes through them in the sorting order and runs the comparisons R.A = S.B. Generally, the attributes A and B can have a complex relationship, i.e., attribute values can appear multiple times in both column R.A and column S.B. In that case, there may be multiple join-compatible tuples from S for any one tuple from R and vice versa. For such attribute values, the respective tuples from R and S must, therefore, be combined via a nested partial join.

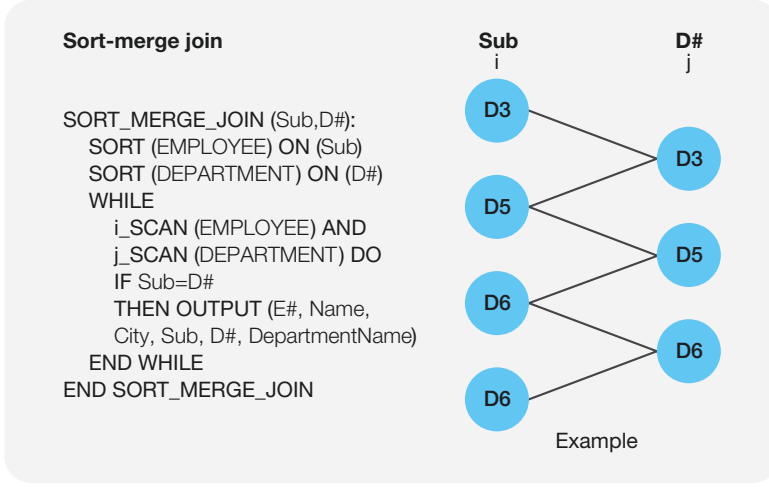


Fig. 5.10 Going through tables in sorting order

In the query of the EMPLOYEE and DEPARTMENT tables, the sort-merge join is linearly dependent on the occurrences of the tuples, since D# is a key attribute. The algorithm only has to go through both tables once to compute the join.

Database systems are generally unable to select a suitable join strategy—or any other access strategy—a priori. Unlike algebraic optimization, this decision hinges on the current content state of the database. It is, therefore, vital that the statistical information contained in the system tables is regularly updated, either automatically at set intervals or manually by database specialists.

5.4 Parallel Processing with MapReduce

Analyses of large amounts of data require a division of tasks utilizing parallelism in order to produce results within a reasonable time. The MapReduce method can be used for both computer networks and mainframes; the following section discusses the first, distributed option.

In a distributed computer network, often consisting of cheap, horizontally scaled components, computing processes can be distributed more easily than data sets. Therefore, the MapReduce method has gained widespread acceptance for web-based search and analysis tasks. It employs parallel processing to generate and sort simple data extracts before outputting the results:

- **Map phase**: Subtasks are distributed between various nodes of the computer network to use parallelism. On the individual nodes, simple key-value pairs are extracted based on a query and then sorted (e.g., via hashing) and output as intermediate results.
- **Reduce phase**: In this phase, the abovementioned intermediate results are consolidated for each key or key range and output as the final result, which consists of a list of keys with the associated aggregated value instances.

Figure 5.11 shows a simple example of a MapReduce procedure. Documents or websites are to be searched for the terms algorithm, database, NoSQL, key, SQL, and distribution. The requested result is the frequency of each term.

The Map phase consists of the two parallel mapping functions M1 and M2. M1 generates a list of key-value pairs, with the search terms as key and their frequencies as value. M2 simultaneously executes a similar search on another computer node with different documents or websites. The preliminary results are then sorted alphabetically with the help of a hashing algorithm. For the upper part, the first letters A to N of the keys (search terms) are the sorting criterion; in the lower part, it is the letters O-Z.

The Reduce phase in Fig. 5.11 combines the intermediate results. The Reduce function R1 adds up the frequencies for the terms starting with A to N; R2 does the same for those starting with O to Z. The result, sorted by frequency of the search terms, are one list with NoSQL (4), database (3), and algorithm (1), and a second list with SQL (3),

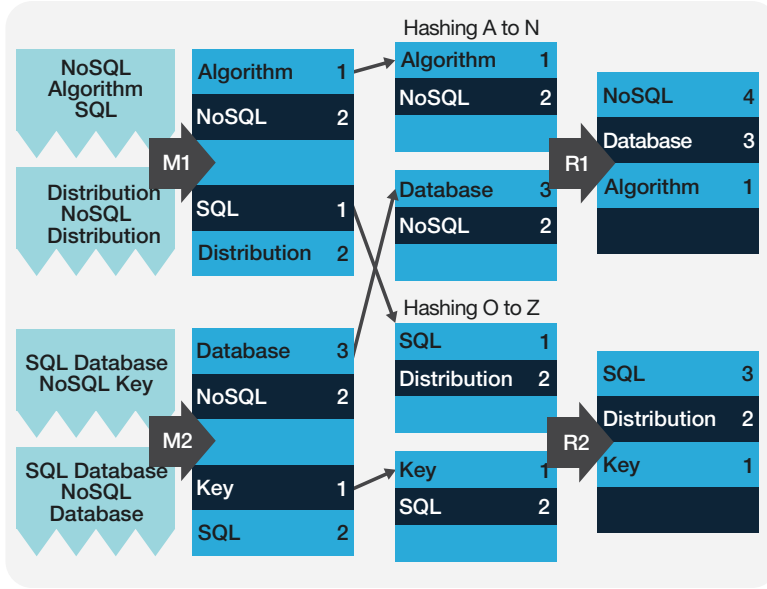


Fig. 5.11 Determining the frequencies of search terms with MapReduce

distribution (2), and key (1). The final result combines these two lists and sorts them by frequency.

The MapReduce method is based on common functional programming languages such as LISP (LISt Processing), where the map() function calculates a modified list for all elements of an original list as an intermediate result. The reduce() function aggregates individual results and reduces them into an output value.

MapReduce has been improved and patented by Google developers for huge amounts of semistructured and unstructured data. However, the function is also available in many open-source tools. The procedure plays an important role in NoSQL databases (Chap. 7), where various manufacturers use the approach for retrieving database entries. Due to its use of parallelism, the MapReduce method is not only useful for data analysis, but also for load distribution, data transfer, distributed searches, categorizations, and monitoring.

5.5 Layered Architecture

It is considered a vital rule for the system architecture of database systems that future changes or expansions must be locally limitable. Similarly to the implementation of operating systems or other software components, *fully independent system layers* that communicate via defined interfaces are introduced into relational and nonrelational database systems.

Figure 5.12 gives an overview of the five layers of system architecture based on relational database technology. The section below further shows how those layers correspond to the major features described in Chap. 4 and the previous sections of Chap. 5.

▶ **Layer 1: Set-oriented Interface** The first layer is used to describe data structures, provide set operations, define access conditions, and check integrity constraints (see Sect. 4.2). Either during early translation and generation of access module or during runtime, it is necessary to check syntax, resolve names, and select access paths. There is room for considerable optimization in the selection of access paths.

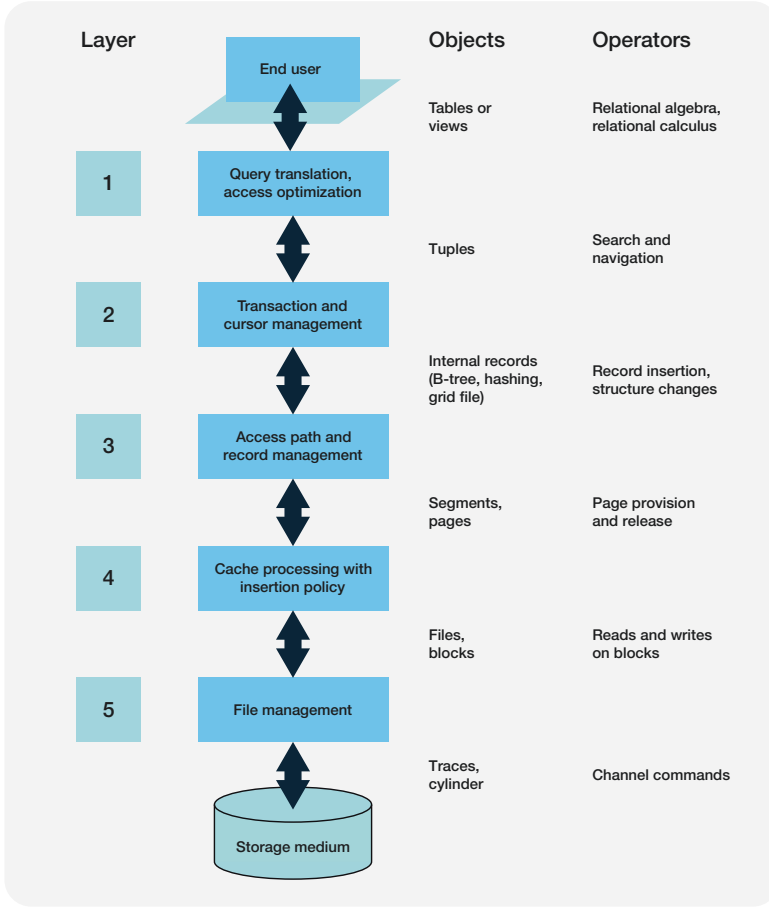


Fig. 5.12 Five-layer model for relational database systems

▶ **Layer 2: Record-oriented Interface** The second layer converts logical records and access paths into physical structures. A cursor concept allows for navigating or processing records according to the physical storage order, positioning specific records within a table, or providing records sorted by value. Transaction management, as described in Sect. 4.2, must be used to ensure that the consistency of the database is maintained and no deadlocks arise between various user requests.

▶ **Layer 3: Storage and access structures** The third layer implements physical records and access paths on pages. The number of page formats is limited, but in addition to tree structures and hashing methods, multidimensional data structures should be supported in the future. These common storage structures are designed for efficient access to external storage media. Physical clustering and multidimensional access paths can also be used to achieve further optimization in record and access path management.

▶ **Layer 4: Page assignment** For reasons of efficiency and to support the implementation of recovery procedures, the fourth layer divides the linear address space into segments with identical page limits. The file management provides pages in a cache on request. On the other hand, pages can be inserted into or substituted within the cache with insertion or replacement policies. There is not only the direct assignment of pages to blocks, but also indirect assignment, such as caching methods which allow for multiple pages to be inserted into the database cache atomically.

▶ **Layer 5: Memory allocation** The fifth layer realizes memory allocation structures and provides block-based file management for the layer above. The hardware properties remain hidden from the file and block-oriented operations. The file management usually supports dynamically growing files with definable block sizes. Ideally, it should also be possible to cluster blocks and input and output multiple blocks with only one operation.

5.6 Use of Different Storage Structures

Many web-based applications use different data storage systems to fit their various services. Using just one database technology, e.g., relational databases, is no longer enough. The wide range of requirements regarding consistency, availability, and partition tolerance demand a mix of storage systems, especially due to the CAP theorem.

Figure 5.13 shows a schematic representation of an online store. In order to guarantee high availability and partition tolerance, session management and shopping carts utilize key-value stores (Chap. 6). Orders are saved to a document store (Chap. 7) and customers and accounts are managed in a relational database system.

Performance management is a vital part of successfully running an online store. Web analytics are used to store key performance indicators (KPIs) of content and visitors in a data warehouse (Chap. 6). Specialized tools such as data mining and predictive business

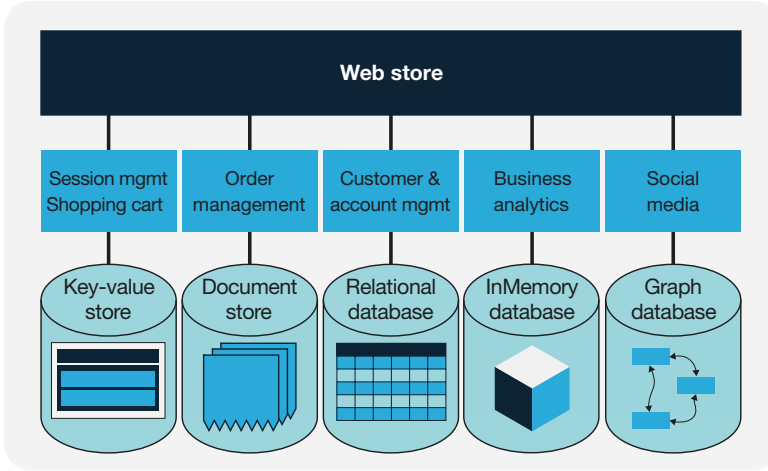


Fig. 5.13 Use of SQL and NoSQL databases in an online store

analysis allow for regular assessments of business goals and the success of campaigns and other actions. Since analyses on a multidimensional data cube are time-consuming, the cube is kept in-memory.

Social media integration for the webshop is a good idea for many reasons. Products and services can be promoted and customers' reactions can be evaluated; in the case of problems or dissatisfaction, good communication and appropriate measures can avoid or mitigate possible negative impacts. Following blogs and relevant discussion threads on social networks can also help to discover and recognize important trends or innovation in the industry. Graph databases (Sect. 7.6) are the logical choice for the analysis of relationships between individual target groups.

The services needed for the online store and the integration of heterogeneous SQL and NoSQL databases can be realized with the REST (Representational State Transfer) architecture. It consists of five elements:

- **Resource identification**: Web resources are identified using a Uniform Resource Identifier (URI). Such resources can, for instance, be websites, files, services, or e-mail addresses. URIs have up to five parts: scheme (type of URI or protocol), authority (provider or server), path, optional query (information to identify a resource), and optional fragment (reference within a resource). An example would be http://eShop.com/customers/12345.
- **Linking**: Resources are connected via hyperlinks, i.e., electronic references. Hyperlinks or simply links are cross-references in hypertext documents that point to a location within the document itself or to another electronic document. An HTML hyperlink looks like this: Browse our online store for contemporary literature .

- **Standard methods**: Any resource on the web can be manipulated with a set of methods. The standard methods of HTTP (HyperText Transfer Protocol), such as GET (request a resource from a server), POST (send data to a server), and DELETE (delete a resource), allow for a unified interface. This ensures that other web services can communicate with all resources at any time.
- **Representations**: Servers based on REST must be able to provide various representations of resources, depending on application and requirements. Besides the standard HTML (HyperText Markup Language) format, resources are often provided in XML (eXtensible Markup Language).
- **Statelessness**: Neither applications nor servers exchange state information between messages. This improves the scalability of services, e.g., load distribution on multiple computer nodes (cf., the MapReduce method).

REST offers a template for the development of distributed applications with heterogeneous SQL and NoSQL components. It ensures horizontal scalability in the case business volumes increase or new services become necessary.

5.7 Further Reading

Some works exclusively discuss the architecture of database systems or even limit it down to individual levels of the layer architecture. Härder (1978) and Härder and Rahm (2001) present basic principles for the implementation of relational database systems (layer model). Lockemann and Schmidt (1993) also spotlight certain aspects of data architecture. Maier (1983), Paredaens et al. (1989), and Ullman (1982) consider the theoretical side of optimization issues.

The standard literature on NoSQL by Celko (2014), Edlich et al. (2011), and Sadalage and Fowler (2013) explains the cornerstones of Big Data and NoSQL, specifically giving an introduction to the MapReduce procedure, the CAP theorem, and to some extent consistent hashing. The book by Redmond and Wilson (2012) provides an insightful exposition of the basics of SQL and NoSQL databases by example of seven specific systems.

In 2010, Google Inc. received a US patent for the MapReduce method for parallel processing of large amounts of data on massive distributed computers. Researchers Dean and Ghemawat (2004) of Google Inc. introduced the procedure at a symposium on operating systems in San Francisco.

Wiederhold (1983) discusses memory structures for database systems. The B-tree was introduced by Bayer (1992), hash functions are described by Maurer and Lewis (1975), and the grid file can be found in Nievergelt et al. (1984). Hash functions are used for multiple purposes in computer science and business IT. Consistent hashing, which places the address space for hash addresses on a circle, was developed mainly for Big Data and especially key-value stores. For example, Amazon uses this technique in their key-value

store Dynamo (DeCandia et al. 2007). MIT researchers Karger et al. (1997) wrote a fundamental paper on consistent hashing and distributed protocols.

The Representational State Transfer or REST paradigm is an architecture proposal for the development of web services. It was put forward by the World Wide Web Consortium (W3C 2014) and is currently used as the basis of most web-based applications. An in-depth work on the subject was published by Tilkov (2011).

References

Bayer, R.: Symmetric Binary B-Trees: Data Structures and Maintenance Algorithms. Acta Inform. **1**(4), 290–306 (1992)

Celko, J.: Joe Celko's Complete Guide to NoSQL—What every SQL Professional needs to know about Nonrelational Databases. Morgan Kaufmann, Amsterdam (2014)

Dean, J., Ghemawat S.: MapReduce—Simplified data processing on large clusters. In: Proc. of the 6th Symposium on Operating Systems, Design and Implementation (OSDI'04), San Francisco, December 6–8, pp. 137–150 (2004)

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo—Amazon's Highly Available Key-value Store. Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP'07), Stevenson, Washington, October 14–17, pp. 205–220 (2007)

Edlich, S., Friedland, A., Hampe, J., Brauer, B., Brückner, M.: NoSQL—Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken. Hanser, München (2011)

Härder, T.: Implementierung von Datenbanksystemen. Hanser, München (1978)

Härder, T., Rahm, E.: Datenbanksysteme—Konzepte und Techniken der Implementierung. Springer, Berlin (2001)

Karger, D., Lehmann, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent Hashing and Random Trees—Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. Proc. of the 29th Annual ACM Symposium on Theory of Computing, El Paso, Texas (1997)

Lockemann, P.C., Schmidt, J.W. (eds.): Datenbank-Handbuch. Springer, Berlin (1993)

Maier, D.: The Theory of Relational Databases. Computer Science Press, Rockville (1983)

Maurer, W.D., Lewis, T.G.: Hash Table Methods. ACM Comput. Surv. **7**(1), 5–19 (1975)

Nievergelt, J., Hinterberger, H., Sevcik, K.C.: The grid file: An adaptable, symmetric multikey file structure. ACM Trans. Database Syst. **9**(1), 38–71 (1984)

Paredaens, J., De Bra, P., Gyssens, M., Van Gucht, D.: The Structure of the Relational Database Model. Springer, Berlin (1989)

Redmond, E., Wilson, J.R.: Seven Databases in Seven Weeks—A Guide to Modern Databases and the NoSQL Movement. The Pragmatic Bookshelf, Dallas (2012)

Sadalage, P.J., Fowler, M.: NoSQL Distilled—A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, Upper Saddle River (2013)

Tilkov, S.: REST und HTTP—Einsatz der Architektur des Web für Integrationsszenarien. dpunkt, Heidelberg (2011)

Ullman, J.: Principles of Database Systems. Computer Science Press, Rockville (1982)

W3C: World Wide Web Consortium. http://www.w3.org/ (2014). Accessed 18 Dec 2014

Wiederhold G.: Database Design. McGraw-Hill, Auckland (1983)

Postrelational Databases

6

6.1 The Limits of SQL—and Beyond

Relational database technology and especially SQL-based databases came to dominate the market in the 1980s and 1990s. Today, SQL databases are still the de facto standard for most database applications in organizations and companies. This time-tested and widely supported technology will in all likelihood continue to be used for the next decades. Nevertheless, the future of databases needs to be discussed. Keywords here are NoSQL databases, graph databases, and distributed database systems, as well as temporal, deductive, semantic, object-oriented, fuzzy, and versioned database systems, etc. What is behind all these terms? This chapter explains some postrelational concepts and shows methods and trends, remaining subjective in its choice of topics. NoSQL databases are described in Chap. 7.

The classic relational model and the corresponding SQL-based database systems admittedly show some disadvantages stemming on the one hand from *extended requirements in new areas of application*, on the other hand from the *functional limits of SQL*. Relational database technology can be applied in a variety of fields and can be seen as the all-rounder among database models. There are, however, niches and scenarios in which SQL-based databases, being transaction and consistency-oriented, are a hindrance, for example when high performance processing of large amounts of data is required. In those cases, the use of specialized tools that are more efficient for the given niche is advisable, even if an SQL database could theoretically be used. Furthermore, there are applications in which SQL and its basis, relational algebra, are simply not sufficient. SQL is a relationally complete language, but it is not Turing complete. There are calculable problems that cannot be solved with relational operations. A major part of this category is the class of recursive problems, such as network analysis with cycles.

A. Meier and M. Kaufmann, *SQL & NoSQL Databases,*
https://doi.org/10.1007/978-3-658-24549-8_6

SQL remains the most important and most popular database language. Today, there is a wide choice of commercial products with enhanced database functionality, some of them open source. It is often not easy for professionals to orientate in the variety of possibilities. Often, the required effort and the economic benefit of a changeover are not clear. Many companies, therefore, still require a considerable amount of mental work to future-proof their application architecture concepts and choose the appropriate product. In a nutshell, concise architecture concepts and migration strategies for the use of postrelational database technologies are still lacking.

In this chapter and in the next, we present a selection of problem cases and possible solutions. Some demands not covered by purely relational databases can be met by individual enhancements of relational database systems, others have to be approached with fundamentally new concepts and methods. Both of these trends are summarized under *postrelational database systems*. We also consider NoSQL postrelational, but cover it in a separate chapter.

6.2 Federated Databases

Noncentralized or *federated databases* are used where data is to be stored, maintained, and processed in different places. A *database is distributed* if the data content is stored on separate computers. Copying all contained data redundantly onto several computers for load balancing is called *replication*. Fragmentation means that for an increased data volume the data is effectively partitioned into smaller parts, so-called fragments, and split between several computers. Fragments are also often called partitions or *shards*, the concept of fragmentation is then accordingly termed partitioning or sharding. A distributed database is *federated* if *several physical data fragments* are kept on *separate computers*, but can be accessed by *one single logical database schema.* The users of a federated database only have to deal with the logical view of the data and can ignore the physical fragments. The database system itself performs the database operations locally or, if necessary, split between several computers.

A simple example of a federated database is shown in Fig. 6.1. Splitting the EMPLOYEE and DEPARTMENT tables into different physical fragments is an important task for the database administrators, not the users. According to our example, the departments IT and HR are geographically based in Cleveland, the accounting department in Cincinnati. Fragment F1 as a partial table of the EMPLOYEE table includes only employees of the IT and the HR departments. Similarly, fragment F2 from the initial DEPARTMENT table shows those departments that are based in Cleveland. Fragments F3 and F4 contain the employees and departments in Cincinnati, respectively.

If a table is split horizontally, keeping the original structure of table rows, the result is called *horizontal fragments*. The individual fragments should not overlap, but combine to form the original table.

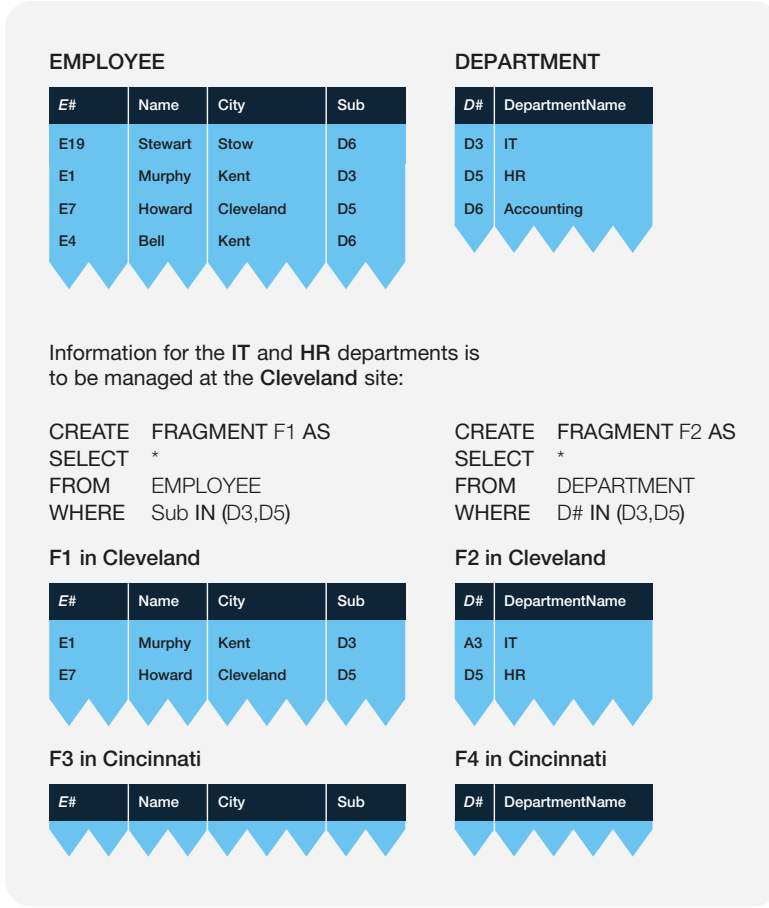


Fig. 6.1 Horizontal fragmentation of the EMPLOYEE and DEPARTMENT tables

Instead of being split horizontally, a table can also be *divided into vertical fragments* by combining several columns along with the identification key, segmenting the tuples. One example is the EMPLOYEE table, where certain parts like salary, qualifications, development potential, etc., would be kept in a vertical fragment restricted to the HR department for confidentiality reasons. The remaining information could be made accessible for the individual departments in another fragment. Hybrid forms between horizontal and vertical fragments are also possible.

One important task of a federated database system is guaranteeing local autonomy. Users can autonomously work with their local data, even if certain computer nodes in the network are unavailable[1].

[1]Periodically extracted parts of tables (called snapshots) improve local autonomy.

Apart from local autonomy, the *principle of noncentralized processing* is of importance. This means the database system can handle queries locally in the different network nodes. For noncentralized applications like these, which demand data from different fragments, the database system has to allow for remote access to read and update tables. In order to achieve this, it has to provide a distributed transaction and recovery concept. These concepts demand special protection mechanisms for distributed databases.

The *internal processing strategy for distributed database queries* is vital here, as the example of querying for employees and department names in Fig. 6.2 illustrates. The query can be formulated in normal SQL without specifying the fragment. The task of the database system is to determine the optimal calculation strategy for this noncentralized query. Both the EMPLOYEE and the DEPARTMENT table are fragmented between Cleveland and Cincinnati. Therefore, *certain calculations are executed locally and synchronously*. Each node organizes the join between the EMPLOYEE and DEPARTMENT

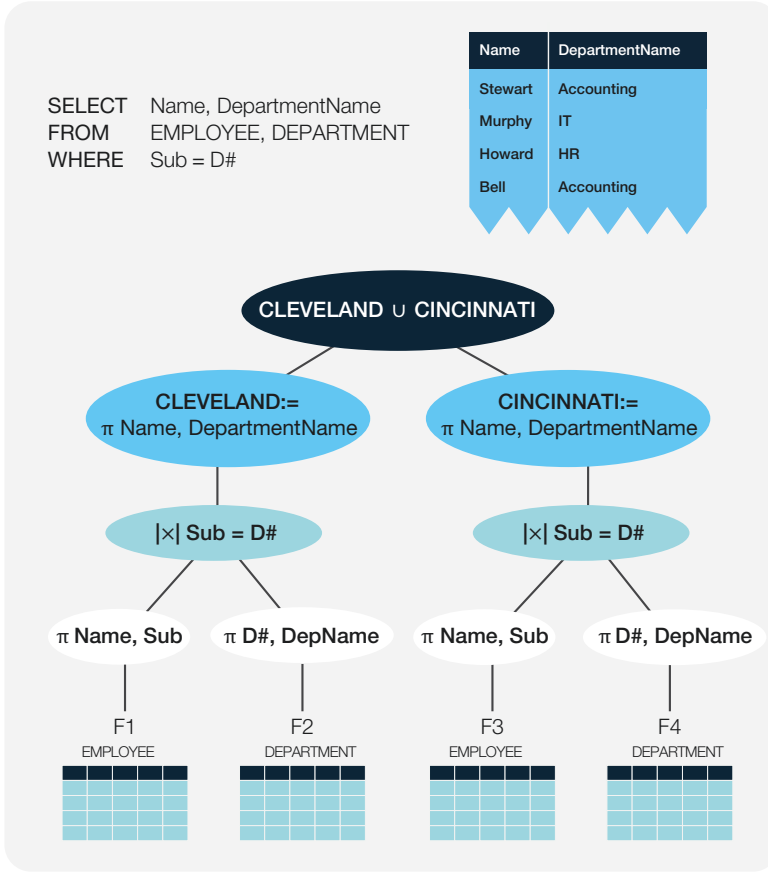


Fig. 6.2 Optimized query tree for a distributed join strategy

fragments independently from the other. After these partial calculations, the final result is formed by a set union of the partial results.

For further optimization, the single nodes make projections on the requested attributes Name and Department Name. Then, the join operations on the reduced table fragments are calculated separately in Cleveland and Cincinnati. Finally, the preliminary results are reduced once more by projecting them on the requested names and department names before a set union is formed.

In calculating noncentralized queries, union and join operations are typically evaluated late in the process. This supports high parallelism in processing and improves performance on noncentralized queries. The maxim of optimization is to put the join operations in the query tree close to the root node, while selections and projections should be placed near the leaves of the query tree.

▶ **Federated database system** A federated database system fulfils the following conditions:

- It supports a single logical database schema and several physical fragments on locally distributed computers.
- It guarantees *transparency regarding to the distribution of databases*, so ad hoc queries and application tools do not have to take into account the physical distribution of the data, i.e., the partitioning.
- It ensures local autonomy, i.e., it allows working locally on its noncentralized data, even if single computer nodes are not available.
- It guarantees the consistency of the distributed databases and internally optimizes the distributed queries and manipulations with a coordination program[2].

The first prototypes of distributed database systems were developed in the early 1980s. Today, relational databases fulfilling the aforementioned demands only partially are available. Moreover, the conflict between partition tolerance and schema integration remains, so that many distributed databases, especially NoSQL databases (Chap. 7), either offer no schema federation, like key-value stores, column family stores, or document stores, or do not support the fragmentation of their data content, like graph databases.

6.3 Temporal Databases

Today's relational database systems are designed to manage information relevant to the present (current information) in tables. For users to query and analyze a relational databases across time, they need to individually manage and update the information relating to the past or future. This is because the database system does not directly support them saving, querying, or analyzing time-related information.

[2]In distributed SQL expressions, the two-phase commit protocol guarantees consistency.

Time is understood as a *one-dimensional physical quantity* whose values are totally ordered so that any two values in the timeline can be compared using the order relations "less than" and "greater than". Not only date and time, such as "April 1, 2016, 2.00 pm" are relevant information, but also durations in the form of time intervals. One example is the age of an employee, determined by a number of years. It is important to note that a given time can be interpreted as either an instant (a point in time) or a time period, depending on the view of the user.

Temporal databases are designed to *relate* data values, individual tuples, or whole tables *to the time axis*. The time specification itself has different meanings for an object in the database, because valid time can either be understood as an instant when a certain event takes place, or as a period if the respective data values are valid throughout a period of time. For instance, the address of an employee is valid until it is next changed.

Another kind of time specification is the transaction time, recording the instant when a certain object is entered into, changed in, or deleted from the database. The database system usually manages the different transaction times itself with the help of a journal, which is why time will always be used in the sense of valid time in the following.

In order to record valid times as points in time, most relational database systems today already support two data types: *DATE is used for dates* in the form year, month, and day, *TIME for the time of the day* in hours, minutes, and seconds. To give a period of time, no special data type has to be chosen; integers and decimals are sufficient. This makes it possible to run calculations on dates and times. One example is the employees table shown in Fig. 6.3, in which Date of Birth and Start Date have been added to the attribute categories. These attributes are time-related, and the system can, therefore,

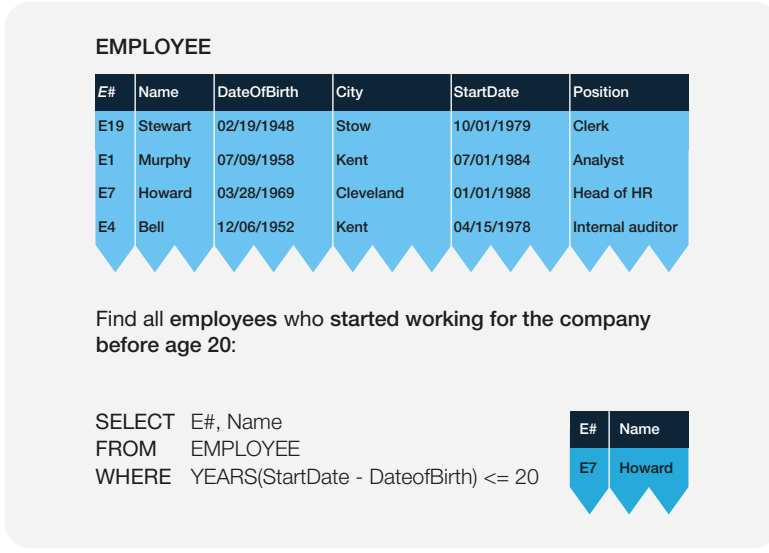


Fig. 6.3 EMPLOYEE table with data type DATE

be queried for a list of all employees who started working for the company before their 20th birthday.

The EMPLOYEE table still offers a snapshot of the current data. Therefore, it is not possible to query into the past nor the future, because there is no information regarding the *valid time* of the data values. If, for instance, the role of employee Howard is changed, the existing data value will be overwritten, and the new role considered as current. However, there is no information as to from and until when employee Howard worked in a specific role.

Two attributes are commonly used to express the *validity of an entity*: The "Valid From" time indicates the point in time when a tuple or a data value became or becomes valid. The attribute "Valid To" indicates the end of a period of validity by giving the corresponding instant. Instead of both the VALID_FROM and VALID_TO times, on the timeline the VALID_FROM instant may be sufficient. The VALID_TO instants are defined implicitly by the following VALID_FROM instants, as the validity intervals of any one entity cannot overlap.

The temporal table TEMP_EMPLOYEE shown in Fig. 6.4 lists all validity statements in the attribute VALID_FROM for the employee tuple M1 (Murphy). This attribute *must* be included in the key so that not only current, but also past and future states can be identified uniquely.

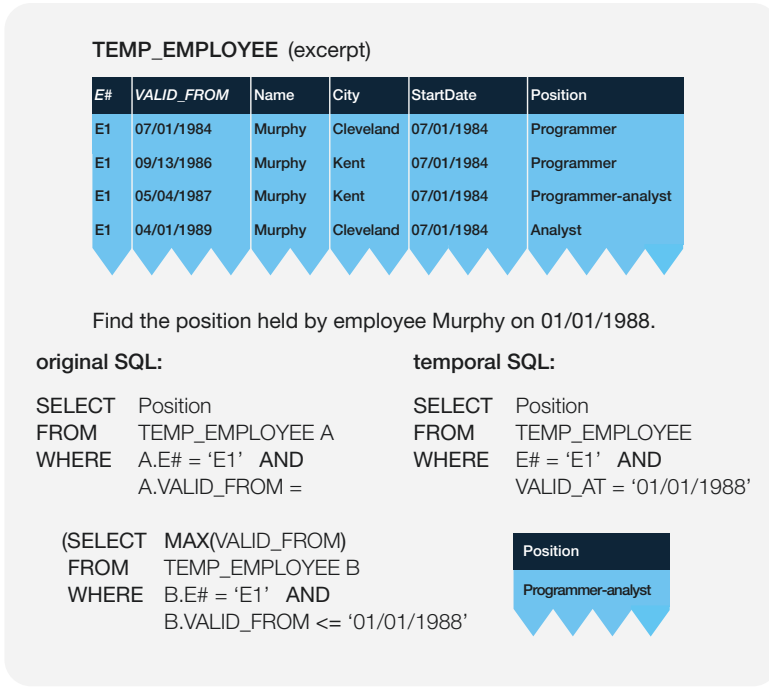


Fig. 6.4 Excerpt from a temporal table TEMP_EMPLOYEE

The four tuples can be interpreted as follows: Employee Murphy used to live in Cleveland from July 1, 1984 to September 12, 1986, then in Kent until March 31, 1989, and has lived in Cleveland again since April 1, 1989. From the day they started working for the company until May 3, 1987, they worked as a programmer, between May 4, 1987 and March 31, 1989 as a programmer-analyst, and since April 1, 1989, they have been working as an analyst. The table TEMP_EMPLOYEE is indeed temporal, as it shows not only current states, but also information about data values related to the past. Specifically, it can answer queries that do not only concern current instants or periods.

For instance, it is possible in Fig. 6.4 to determine the role employee Murphy had on January 1st 1988. Instead of the original SQL expression of a nested query with the MAX function (Sect. 3.3.1 and SQL tutorial), a language directly supporting temporal queries is conceivable. The keyword VALID_AT determines the time for which all valid entries are to be queried.

Temporal database system A temporal database management system (TDBMS)

- supports the time axis as valid time by ordering attribute values or tuples by time and
- contains temporal language elements for queries into future, present, and past.

In the field of temporal databases there are several language models facilitating work with time-related information. Especially the operators of relational algebra and relational calculus have to be expanded in order to enable a join of temporal tables. The rules of referential integrity also need to be adapted and interpreted as relating to time. Even though these kinds of methods and corresponding language extensions have already proven themselves in R&D, very few database systems today support temporal concepts. The SQL standard also supports temporal databases.

6.4 Multidimensional Databases

Operative databases and applications are focused on a clearly defined, function-oriented performance area. Transactions aim to provide data for business handling as quickly and precisely as possible. This kind of business activity is often called *online transaction processing* or OLTP. Since the operative data has to be overwritten daily, users lose important information for decision-making. Furthermore, these databases were designed primarily for day-to-day business, not for analysis and evaluation. Recent years have, therefore, seen the development of specialized databases and applications for data analysis and decision support, in addition to transaction-oriented databases. This process is termed *online analytical processing* or OLAP.

At the core of OLAP is a *multidimensional database*, where all decision-relevant information can be stored according to various analysis dimensions (data cube). Such databases can become rather large, as they contain decision-making factors from

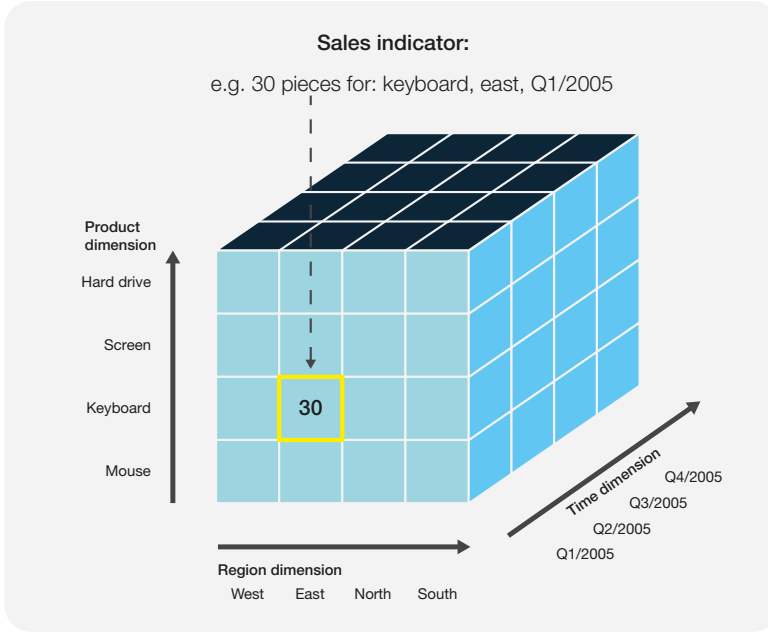


Fig. 6.5 Data cube with different analysis dimensions

multiple points in time. Sales figures, for instance, can be stored and analyzed in a multidimensional database by quarter, region, or product.

This is demonstrated in Fig. 6.5, which also illustrates the concept of a multidimensional database. It shows the three analysis dimensions product, region, and time. The term dimension describes the axes of the data cube. The design of the dimensions is important, since analyses are executed along these axes. The order of the dimensions does not matter, every user can and should analyze the data from their own perspective. Product managers, for instance, prioritize the product dimension; salespeople prefer sales figures to be sorted by region.

The dimensions themselves can be structured further. The product dimension can contain product groups; the time dimension could cover not only quarters, but also days, weeks, and months. A dimension, therefore, also describes the desired aggregation levels valid for the data cube.

From a logical point of view, in a multidimensional database or a data cube it is necessary to specify not only the dimensions, but also the indicators[3]. An *indicator* is a key figure or parameter needed for decision support. These key figures are aggregated by

[3]Indicators are often also called facts, see also Sect. 6.7 on facts and rules of knowledge databases.

analysis and grouped according to the dimension values. Indicators can relate to quantitative as well as qualitative characteristics of the business. Apart from financial key figures, meaningful indicators concern the market and sales, customer base and customer fluctuation, business processes, innovation potential, and know-how of the employees. Indicators, in addition to dimensions, are the basis for the management's decision support, internal and external reporting, and a computer-based performance measurement system.

The main characteristic of a star schema is the classification of data as either indicator data or dimension data. Both groups are shown as tables in Fig. 6.6. The indicator table is at the center, the descriptive dimension tables are placed around it; one table per dimension. The dimension tables are attached to the indicator table forming a star-like structure.

Should one or more dimensions be structured, the respective dimension table could have other dimension tables attached to it. The resulting structure is a *snowflake schema* showing aggregation levels of the individual dimensions. In Fig. 6.6, for instance, the time dimension table for the first quarter of 2005 could have another dimension table attached, listing the calendar days from January to March 2005. Should the dimension month be necessary for analysis, a month dimension table would be defined and connected to the day dimension table.

The classic relational model can be used for the implementation of a multidimensional database. Figure 6.7 shows how indicator and dimension tables of a star schema are implemented. The indicator table is represented by the relation F_SALES, which has a multidimensional key. This concatenated key needs to contain the keys for the dimension tables D_PRODUCT, D_REGION, and D_TIME. In order to determine sales lead

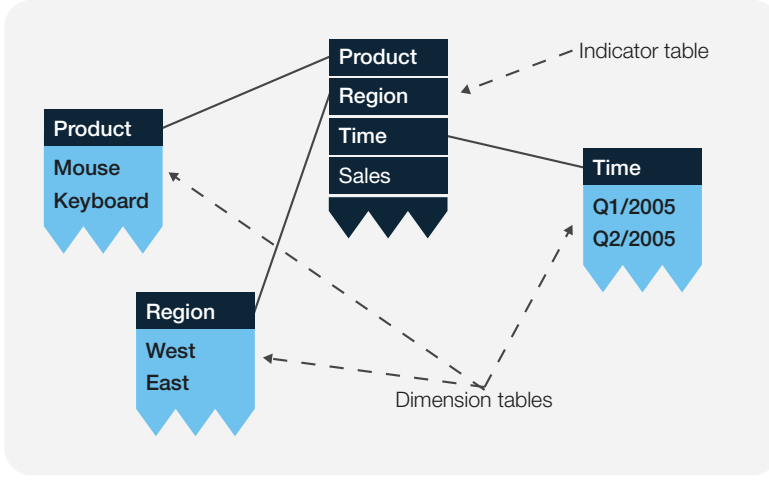


Fig. 6.6 Star schema for a multidimensional database

Find the Apple sales for the 1. quarter of 2005 by sales lead Mahoney.

```
SELECT   SUM(Revenue)
FROM     D_PRODUCT D1, D_REGION D2, D_TIME D3, F_SALES F
WHERE    D1.P# = F.P# AND
         D2.R# = F.R# AND
         D3.T# = F.T# AND
         D1.Supplier = 'Apple' AND
         D2.SalesLead = 'Mahoney' AND
         D3.Year = 2005 AND
         D3.Quarter = 1
```

D_TIME

T#	Year	Quarter
T1	2005	1

F_SALES

P#	R#	T#	Quantity	Revenue
P2	R2	T1	30	160,000

D_PRODUCT

P#	ProductName	Supplier
P2	Keyboard	Apple

D_REGION

R#	Name	SalesLead
R2	East	Mahoney

Fig. 6.7 Implementation of a star schema using the relational model

Mahoney's revenue on Apple devices in the first quarter of 2005, it is necessary to formulate a complicated join of all involved indicator and dimension tables (see SQL statement in Fig. 6.7).

A relational database system reaches its limits when faced with extensive multidimensional databases. Formulating queries with a star schema is also complicated and prone to error. There are multiple other disadvantages when working with the classic relational model and conventional SQL. In order to aggregate several levels, a star schema has to be expanded into a snowflake schema, and the resulting physical tables further impair the response time behavior. If users of a multidimensional database want to query more details for deeper analysis (drill-down) or analyze further aggregation levels (roll-up), conventional SQL will be of no use. Moreover, extracting or rotating parts of the data cube, as commonly done for analysis, requires specific software or even hardware. Because of these shortcomings, some providers of database products have decided to add appropriate tools for these purposes to their software range. In addition, the SQL standard has been extended on the language level in order to simplify the formulation of cube operations, including aggregations.

▶ **Multidimensional database system** A multidimensional database management system (MDBMS) supports a data cube with the following functionality:

- For the design, several dimension tables with arbitrary *aggregation levels* can be defined, especially for the time dimension.
- The analysis language offers functions for drill-down and roll-up.
- Slicing, dicing, and rotation of data cubes are supported.

Multidimensional databases are often the core of data warehouses. Unlike multidimensional databases alone, a data warehouse as a distributed information system combines mechanisms for integration, historicization, and analysis of data across several applications of a company, along with processes for decision-support and the management and development of data flows within the organization.

6.5 Data Warehouse

The more and more easily digital data is available, the greater the need to analyze this data for decision support. The management of a company is supposed to base their decisions on facts that can be gathered from the analysis of the available data. This process is called *business intelligence*. Due to heterogeneity, volatility, and fragmentation of the data, cross-application data analysis is often complex: Data is stored heterogeneously in several databases in an organization. Additionally, often only the current version is available. In the source systems, data from one larger subject area, like customers or contracts, is rarely available in one place, but has to be gathered, or integrated, via various interfaces. Furthermore, this data distributed among many databases needs to be sorted into timelines for various subject areas, each spanning several years. Business Intelligence, therefore, makes three demands on the data to be analyzed:

- Integration of heterogeneous data
- Historicization of current and volatile data
- Complete availability of data on certain subject areas.

The three previously introduced postrelational database systems basically cover one of those demands each: The integration of data can be carried out with federated database systems with a central logical schema, historicization of data is possible with temporal databases, and multidimensional databases can provide data on various subject areas (dimensions) for analysis.

As relational database technology has become so wide-spread in practice, the properties of distributed, temporal, and multidimensional databases can be simulated quite well with regular multidimensional databases and some software enhancements. The concept

of *data warehousing* implements these aspects of federated, temporal, and multidimensional database systems using conventional technologies.

In addition to those three aspects, however, there is the demand of *decision support*. Organizations need to analyze data as timelines, so that the complete data on any subject area is available in one place. However, as data in larger organizations is spread among a number of databases, a concept[4] to prepare it for analysis and utilization is necessary.

Data warehouse A data warehouse or DWH is a distributed information system with the following properties:

- *Integrated:* Data from various sources and applications (source systems) is periodically integrated[5] and filed in a uniform schema.
- *Read only:* Data in the data warehouse is not changed once it is written.
- *Historicized:* Thanks to a time axis, data can be evaluated for different points in time.
- *Analysis-oriented:* All data on different subject areas like customers, contracts, or products is fully available in one place.
- *Decision support:* The information in data cubes serves as a basis for management decisions.

A data warehouse offers parts of the functionalities of federated, temporal, and multidimensional databases. Additionally, there are programmable loading scripts as well as specific analysis and aggregation functions. Based on distributed and heterogeneous data sources, business-relevant facts need to be available in such a way that they can efficiently and effectively be used for decision support and management purposes.

Data warehouses can integrate various internal and external data sets (data sources). The aim is to be able to store and analyze, for various business purposes, a consistent and historicized set of data on the information scattered across the company. To this end, data from many sources is integrated into the data warehouse via interfaces and stored there, often for years. Building on this, data analyses can be carried out to be presented to decision-makers and used in business processes. Furthermore, business intelligence as a process has to be controlled by management.

The individual steps of data warehousing are summarized in the following paragraphs (Fig. 6.8).

The *data* of an organization is distributed across several source systems, for instance web platforms, accounting (enterprise resource planning, ERP), and customer databases (customer relationship management, CRM). In order to analyze and relate this data, it needs to be integrated.

[4]For more information, look up the KDD (knowledge discovery in databases) process.

[5]See the ETL (extract, transform, and load) process below.

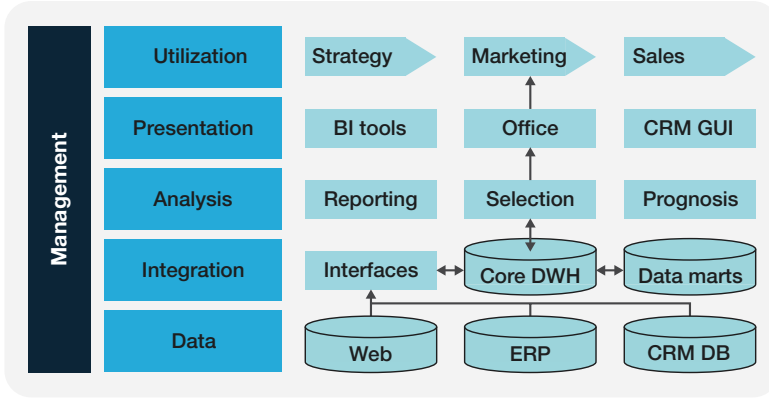


Fig. 6.8 Data warehouse in the context of business intelligence processes

For this *integration* of the data, an ETL (extract, transform, load) process is necessary. The corresponding *interfaces* usually transfer data in the evening or on weekends, when the IT system is not needed by the users. High performance systems today feature continuous loading processes, feeding data 24/7 (*trickle feed*). When updating a data warehouse, *periodicity* is taken into account, so users can see how *up-to-date* their evaluation data is. The more frequently the interfaces load data into the data warehouse, the more up-to-date is the evaluation data. The aim of this integration is *historicization*, i.e., the creation of a timeline in one logically central storage location. The core of a data warehouse (Core DWH) is often modeled in second or third normal form. Historicization is achieved using validity statements (valid_from, valid_to) in additional columns of the tables, as described in Sect. 6.3 on temporal databases. In order to make the evaluation data sorted by subject available for OLAP analysis, individual subject areas are loaded into data marts, which are often realized multidimensionally with star schemas.

The data warehouse exclusively serves for the *analysis of data*. The dimension of time is an important part of such data collections, allowing for more meaningful statistical results. Periodic reporting produces lists of key performance indicators. Data mining tools like classification, selection, prognosis, and knowledge acquisition use data from the data warehouse in order to, for instance, analyze customer or purchasing behavior and utilize the results for optimization purposes. In order for the data to generate value, the insights including the results of the analysis need to be communicated to the decision-makers and stakeholders. The respective analyses or corresponding graphics are made available using a range of interfaces of business intelligence tools (BI tools) or graphical user interfaces (GUI) for office automation and customer relationship management. Decision-makers can utilize the analysis results from data warehousing *in business processes* as well as in strategy, marketing, and sales.

6.6 Object-Relational Databases

In order to store information on books in a relational database, several tables need to be defined, three of which are shown in Fig. 6.9. In the BOOK table, every book has the attributes Title and Publisher added.

Since a book can have more than one author and, reversely, an author can have published multiple books, every author involved is listed in an additional AUTHOR table.

The attribute Name is not fully functionally dependent on the combined key of the Author and Book Number, which is why the table is neither in the second nor any

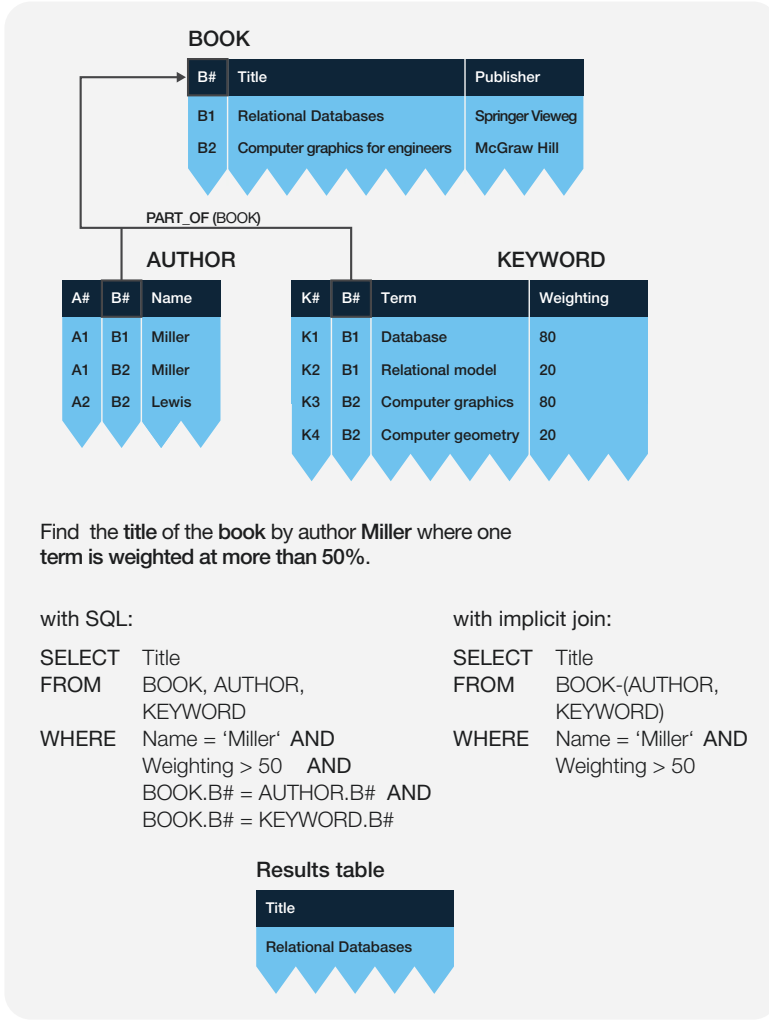


Fig. 6.9 Query of a structured object with and without implicit join operator

higher normal form. The same holds true for the KEYWORD table, because there is a complex-complex relationship between books and their keywords. Weighting is a typical relationship attribute, Label, however, is not fully functionally dependent on the Keyword Number and Book Number key. For proper normalization, the management of books would, therefore, require several tables, since in addition to the relationship tables AUTHOR and KEYWORD, separate tables for the attributes Author and Keyword would be necessary. A relational database would certainly also include information on the publisher in a separate PUBLISHER table, ideally complemented by a table for the relationship between BOOK and PUBLISHER.

Splitting the information about a book between different tables has its disadvantages and is hardly understandable from the point of view of the users, who want to find the attributes of a certain book well-structured in a single table. The relational query and data manipulation language should serve to manage the book information using simple operators. There are also performance disadvantages if the database system has to search various tables and calculate time-consuming join operators in order to find a certain book. To mitigate these problems, extensions to the relational model have been suggested.

A first extension of the relational database technology is to *explicitly declare structural properties to the database system*, for instance by assigning *surrogates*. A surrogate is a *permanent, invariant key value defined by the system*, which uniquely identifies every tuple in the database. Surrogates, as invariant values, can be used to define system-controlled relationships even in different places within a relational database. They support referential integrity as well as generalization and aggregation structures.

In the BOOK table in Fig. 6.9, the book number B# is defined as a surrogate. This number is used again in the dependent tables AUTHOR and KEYWORD under the indication PART_OF(BOOK) (see rule 7 for aggregation in Sect. 2.3.2). Because of this reference, the database system explicitly recognizes the structural properties of the book, author, and keyword information and is able to use them in database queries, given that the query and manipulation language is extended accordingly. An example for this is the implicit hierarchical join operator in the FROM clause that connects the partial tables AUTHOR and KEYWORD belonging to the BOOK table. It is not necessary to state the join predicates in the WHERE clause, as those are already known to the database system through the explicit definition of the PART_OF structure.

Storage structures can be implemented more efficiently by introducing to the database system a PART_OF or analogously an IS_A structure (Sect. 2.2.3). This means that the logical view of the three tables BOOK, AUTHOR, and KEYWORD is kept, while the book information is physically stored as structured objects[6] so that a single database access makes it possible to find a book. The regular view of the tables is kept, and the individual tables of the aggregation can be queried as before.

[6]Research literature also calls them “complex objects”.

BOOK_OBJECT

B#	Title	Publisher	Autor		Keyword		
			A#	Name	K#	Term	Wgt.
B1	Relational...	Springer Vieweg	A1	Miller	K1	Database	80
					K2	Relational model	20
B2	Computer...	McGraw Hill	A1	Miller	K3	Comp. graphics	50
			A2	Lewis	K4	Comp. geometry	50

Fig. 6.10 BOOK_OBJECT table with attributes of the relation type

Another possibility for the management of structured information is *giving up the first normal form*[7] and allowing tables as attributes. Figure 6.10 illustrates this with an example presenting information on books, authors, and keywords in a table. This also shows an object-relational approach, managing a book as one object in the single table BOOK_OBJECT. An object-relational database system can explicitly incorporate structural properties and offer operators for objects and parts of objects.

A database system is *structurally object-relational* if it supports structured object types as shown in Fig. 6.10. In addition to object identification, structure description, and the availability of generic operators (methods like implicit join, etc.), a fully object-relational database system should support the definition of new object types (classes) and methods. Users should be able to determine the methods necessary for an individual object type themselves. They should also be able to rely on the support of inherited properties so that they do not have to define all new object types and methods from scratch, but can draw on already existing concepts.

Object-relational database systems make it possible to treat structured objects as units and use fitting generic operators with them. The formation of classes using PART_OF and IS_A structures is allowed and supported by methods for saving, querying, and manipulating.

▶ **Object-relational database system** An object-relational database management system (ORDBMS) can be described as follows:

- It allows the *definition of object types* (often called classes in reference to object-oriented programming), which themselves can consist of other object types.
- Every database object can be structured and identified through surrogates.

[7]The NF2 model (NF2 = non first normal form) supports nested tables.

- It supports *generic operators* (methods) affecting objects or parts of objects, while their internal representation remains invisible from the outside (data encapsulation).
- Properties of objects can be inherited. This *property inheritance* includes the structure and the related operators.

The SQL standard has for some years been supporting certain object-relational enhancements: object identifications (surrogates); predefined data types for set, list, and field; general abstract data types with the possibility of encapsulation; parametrizable types; type and table hierarchies with multiple inheritance; and user-defined functions (methods).

Object-relational mapping

Most modern programming languages are object-oriented; at the same time, the majority of the database systems used are relational. Instead of migrating to object-relational or even object-oriented databases, which would be rather costly, objects and relations can be mapped to each other during software development if relational data is accessed with object-oriented languages. This concept of *object-relational mapping (ORM)* is illustrated in Fig. 6.11. In this example, there is a relational database management system (RDBMS) with a table AUTHOR, a table BOOK, and a relationship table AUTHORED, since there is a complex-complex relationship (Sect. 2.2.2) between books and authors.

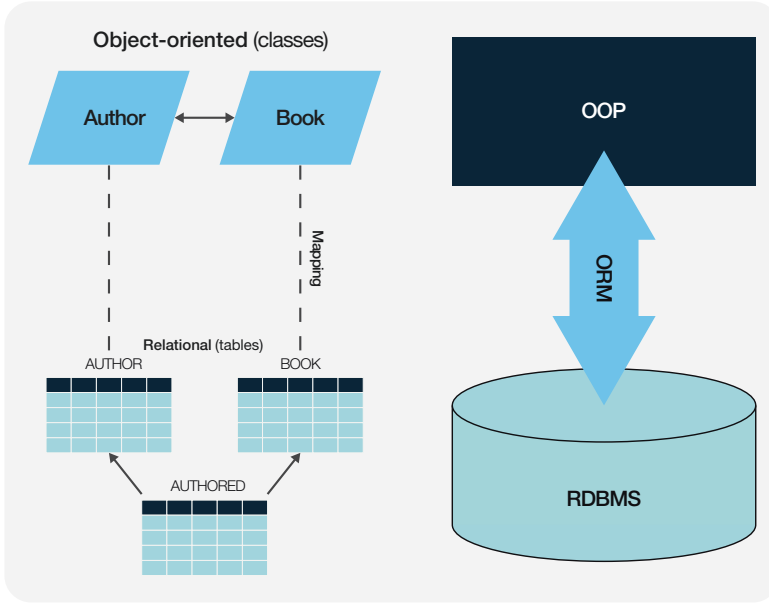


Fig. 6.11 Object-relational mapping

The data in those tables is to be used directly as classes in software development in a project with object-oriented programming (OOP).

An ORM software can automatically map classes to tables, so for the developers it seems as if they are working with object-oriented classes, even though the data is saved in database tables in the background. The programming objects in the main memory are thus persistently written, i.e., saved to permanent memory.

In Fig. 6.11, the ORM software provides the two classes Author and Book for the tables AUTHOR and BOOK. For each line in the table, there is one object as an instance of the respective class. The relationship table AUTHORED is not shown as a class: object-orientation allows for the use of nonatomic object references; thus, the set of books the author has written is saved in a vector field `books[]` in the Author object, the group of authors responsible for a book are shown in the field `authors[]` in the Book object.

The use of ORM is simple. The ORM software automatically derives the corresponding classes based on existing database tables. Records from these tables can then be used as objects in software development. ORM is, therefore, one possible way towards object-orientation with which the underlying relational database technology can be retained.

6.7 Knowledge Databases

Knowledge databases or *deductive databases* cannot only manage the actual data—called *facts*—but also *rules*, which are used to deduct new table contents or facts.

The EMPLOYEE table in Fig. 6.12 is limited to the names of the employees for simplicity. It is possible to define facts or statements on the information in the table, in this case on the employees. Generally, facts are statements that *unconditionally take the truth value TRUE.* For instance, it is true that Howard is an employee. This is expressed by the fact "is_employee (Howard)". For the employees' direct supervisors, a new SUPERVISOR table can be created, showing the names of the direct supervisors and the employees reporting to them as a pair per tuple. Accordingly, facts "is_supervisor_of (A,B)" are formulated to express that "A is a direct supervisor of B".

The job hierarchy is illustrated in a tree in Fig. 6.13. Looking for the direct supervisor of employee Murphy, the SQL query analyzes the SUPERVISOR table and finds supervisor Howard. Using a logic query language (inspired by Prolog) yields the same result.

Besides actual facts, it is possible to define *rules for the deduction of unknown table contents*. In the relational model, this is called a *derived relation* or *deduced relation*. Simple examples of a derived relation and the corresponding derivation rule are given in Fig. 6.14. It shows how the supervisor's supervisor for every employee can be found. This may, for instance, come in useful for large companies or businesses with remote branches in the case when the direct supervisor of an employee is absent and the next higher level needs to be contacted via e-mail.

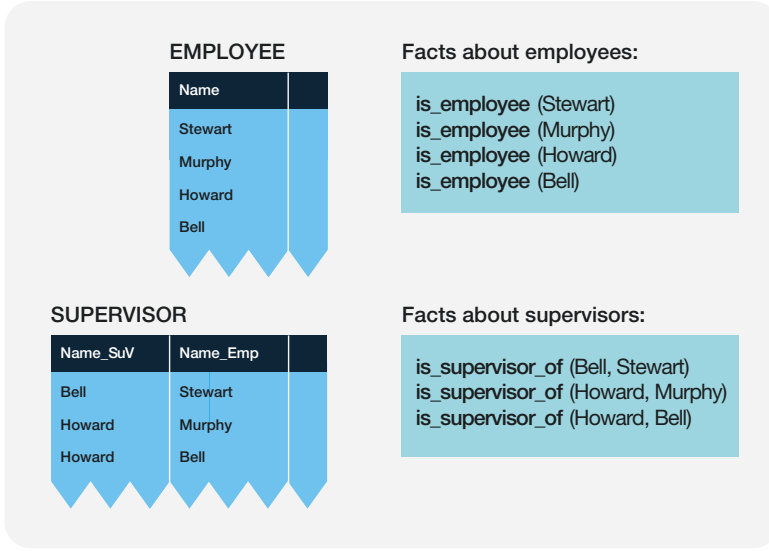


Fig. 6.12 Comparison of tables and facts

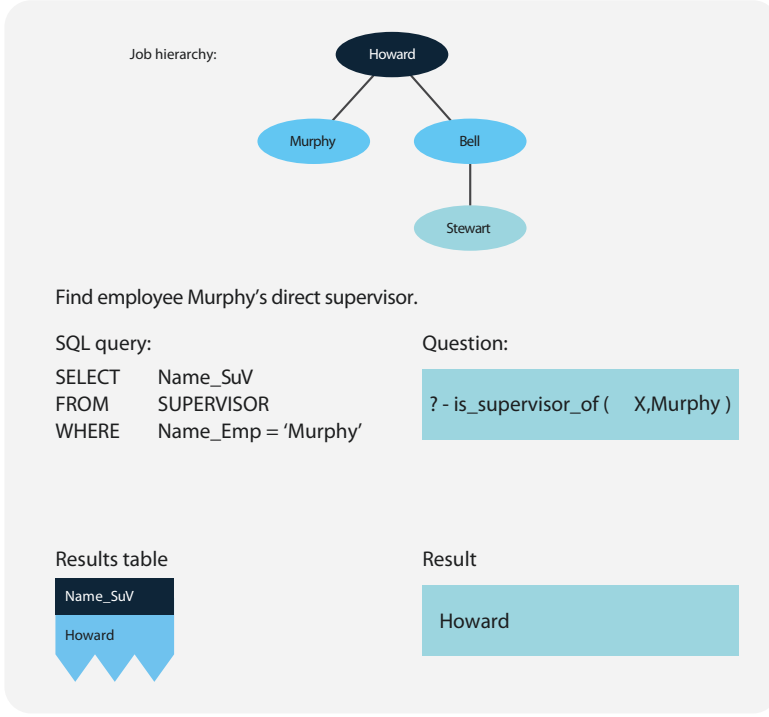


Fig. 6.13 Analyzing tables and facts

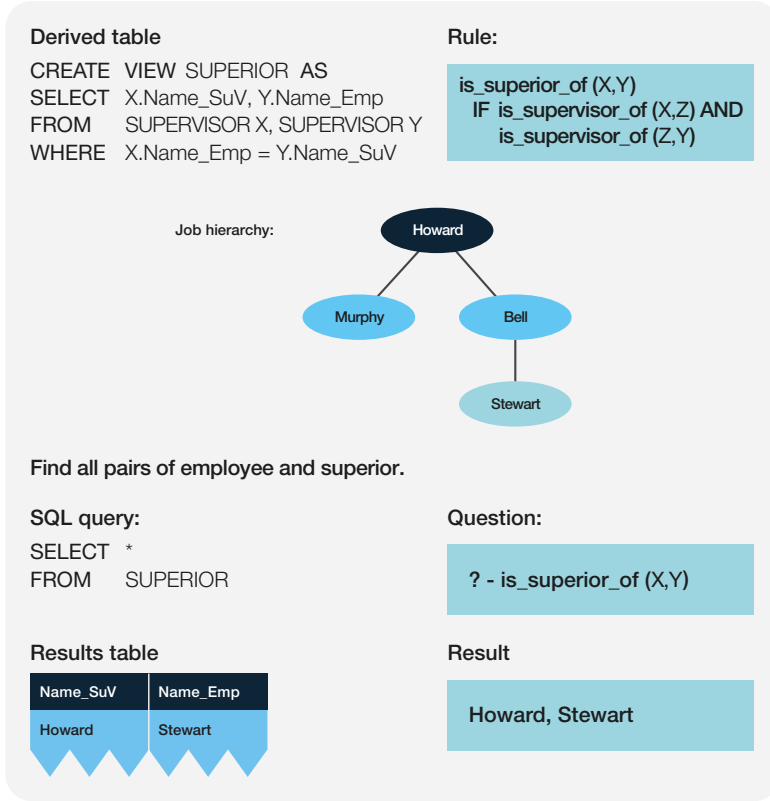


Fig. 6.14 Derivation of new information

The definition of a derived relation corresponds to the definition of a view. In the given example, such a view with the name SUPERIOR is used to determine the next-but-one supervisor of any employee, formed by a join of the SUPERVISOR table with itself. A derivation rule can be defined for this view. The rule "is_superior_of (X,Y)" results from there being a Z where X is the direct supervisor of Z and Z in turn is the direct supervisor of Y. This expresses that X is the next-but-one supervisor of Y, because Z is between them.

A database equipped with facts and rules automatically becomes a *method or knowledge base,* as it does not only contain obvious facts like "Howard is an employee" or "Howard is the direct supervisor of Murphy and Bell", but also derived findings like "Howard is superior supervisor of Stewart". In order to find superior supervisors, the view SUPERIOR defined in Fig. 6.14 is used. The SQL query of this view results in a table with the information that there is only one relationship with a superior supervisor, specifically employee Stewart and their superior supervisor Howard. Applying the corresponding derivation rule "is_superior_of" yields the same result.

A deductive database as a vessel for facts and rules also supports the *principle of recursion*, making it possible to draw an unlimited amount of correct conclusions due to the rules included in the deductive database. Any true statement always leads to new statements.

The principle of recursion can refer to either the *objects in the database or the derivation rules*. Objects defined as recursive are structures that themselves consist of structures and, similar to the abstraction concepts of generalization and aggregation, can be understood as hierarchical or network-like object structures. Furthermore, statements can be determined recursively; in the company hierarchy example all direct and indirect supervisor relationships can be derived from the facts "is_employee" and "is_supervisor_of".

The calculation process which derives all transitively dependent tuples from a table forms the transitive closure of the table. This operator does not belong to the original operators of relational algebra; rather, the transitive closure is a natural extension of the relational operators. It cannot be formed with a fixed number of calculation steps, but only by several relational join, projection, and union operators, whose number depends on the content of the table in question.

These explanations can be condensed into the following definition:

Knowledge database systems A knowledge database management system (KDBMS) supports deductive databases or knowledge bases if

- it contains not only data, i.e., *facts*, but also *rules;*
- the *derivation component* allows for further facts to be derived from facts and rules; and
- it supports *recursion*, which, among other things, allows to calculate the transitive closure of a table.

An *expert system* is an information system that provides specialist knowledge and conclusions for a certain limited field of application. Important components are a knowledge base with facts and rules, and a derivation component for the derivation of new findings. The fields of databases, programming languages, and artificial intelligence will increasingly influence each other and in the future provide efficient problem-solving processes for practical application.

6.8 Fuzzy Databases

Conventional database systems assume attribute values to be precise, certain, and crisp, and queries deliver clear results:

- The attribute values in the databases are *precise*, i.e., they are unambiguous. The first normal form demands attribute values to be atomic and come from a well-defined domain. Vague attribute values, such as "2 or 3 or 4 days" or "roughly 3 days" for the delivery delay of supplier, are not permitted.
- The attribute values saved in a relational database are *certain*, i.e., the individual values are known and, therefore, true. An exception are NULL values, i.e., attribute values that are not known or not yet known. Apart from that, database systems do not offer modeling components for existing uncertainties. Probability distributions for attribute values are, therefore, impossible; expressing whether or not an attribute value corresponds to the true value remains difficult.
- Queries to the database are *crisp*. They always have a binary character, i.e., a query value specified in the query must either be identical or not identical with the attribute values. Querying a database with a query value "more or less" identical with the stored attribute values is not allowed.

In recent years, discoveries from the field of *fuzzy logic* have been applied to data modeling and databases. Permitting incomplete or vague information opens a wider field of application. Most of these works are theoretical; however, some research groups are trying to demonstrate the usefulness of fuzzy database models and database systems with implementations.

The approach shown here is based on the *context model* to define classes of data sets in the relational database schema. There are crisp and fuzzy classification methods. For a crisp classification, database objects are binarily assigned to a class, i.e., the membership function of an object to a class is 0 for "not included" or 1 for "included." A conventional process would, therefore, group a customer either into the class "Customers with revenue problems" or into the class "Customers to expand business with." A fuzzy process, however, allows for membership function values between 0 and 1. A customer can belong in the "Customers with revenue problems" class with a value of 0.3 and at the same time in the "Customers to expand business with" class with a value of 0.7. A fuzzy classification, therefore, allows for a *more differentiated interpretation of class membership*: Database objects can be distinguished between border and core objects, additionally database objects can belong to two or more different classes at the same time.

In the fuzzy-relational database model with contexts, context model for short, every attribute A_j defined on a domain $D(A_j)$ has a context assigned. *A context $K(A_j)$ is a partition of $D(A_j)$ into equivalence classes.* A relational database schema with contexts, therefore, consists of a set of attributes $A=(A_1,\ldots,A_n)$ and another set of associated contexts $K=(K_1(A_1),\ldots,K_n(A_n))$.

For the assessment of customers, revenue and loyalty are used as an example. Additionally, those qualifying attributes are split into two equivalence classes each. The associated attributes and contexts for the customer relationship management are:

- Revenue in dollar per month: The domain for revenue in dollars is defined as [0…1000]. Two equivalence classes [0…499] for small revenues and [500…1000] for large revenues are also created.
- Customer loyalty: The domain {bad, weak, good, great} supplies the values for the Customer loyalty attribute. It is split further into the equivalence classes {bad, weak} for negative loyalty and {good, great} for positive loyalty.

The suggested attributes with their equivalence classes show an example of a numeric and a qualitative attribute each. The respective contexts are:

- K(revenue) = {[0…499], [500…1000] }
- K(loyalty) = {{bad, weak}, {good, great} }

The partitioning of the revenue and loyalty domains results in the four equivalence classes C1, C2, C3, and C4 shown in Fig. 6.15. The meaning of the classes is expressed by semantic class names; for instance, customers with little revenue and weak loyalty are labeled "Don't invest" in C4; C1 could stand for "Retain customer", C2 for "Improve loyalty", and C3 for "Increase revenue". It is the database administrators' job, in cooperation with the marketing department, to define the attributes and equivalence classes and to specify them as an extension of the database schema.

Customer relationship management aims to take into account the customers' individual wishes and behavior instead of only focusing on product-related arguments and efforts. If customers are seen as an asset (*customer value*), they have to be treated according to their market and resource potential. With sharply divided classes, i.e., traditional customer segments, this is hardly possible, as all customers of one class are treated

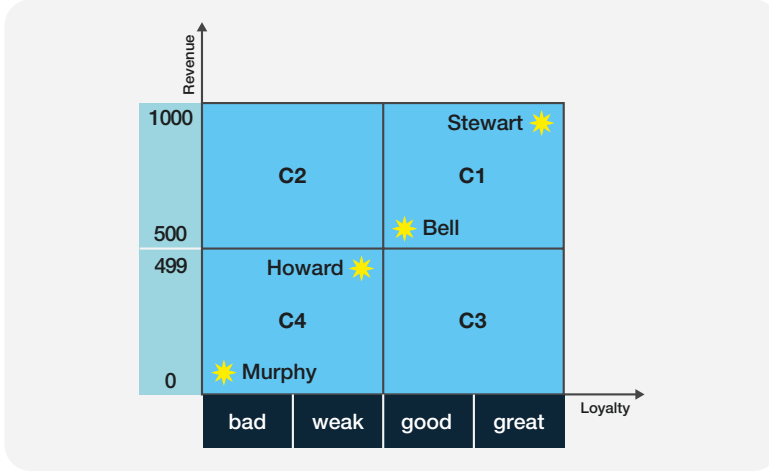


Fig. 6.15 Classification matrix with the attributes Revenue and Loyalty

the same. In Fig. 6.15, for instance, Bell and Howard have almost the same revenue and loyalty. Nevertheless, in a sharp segmentation they are classed differently: Bell falls into the premium class C1 (Retain customer) and Howard into the loser class C4 (Don't invest). Additionally, top customer Stewart is treated the same as Bell, since both belong into segment C1.

As can be seen in Fig. 6.15, the following conflicts can arise from sharp customer segmentation:

- Customer Bell has barely any incentives to increase revenue or loyalty. They belong to the premium class C1 and enjoy the corresponding advantages.
- Customer Bell could face an unpleasant surprise, should their revenue drop slightly or their loyalty rating be reduced. They may suddenly find themselves in a different customer segment; in an extreme case they could drop from the premium class C1 into the loser class C4.
- Customer Howard has a robust revenue and medium customer loyalty, but is treated as a loser. It would hardly be surprising if Howard investigated their options on the market and moved on.
- A sharp customer segmentation also creates a critical situation for customer Stewart. They are, at the moment, the most profitable customer with an excellent reputation, yet the company does not recognize and treat them according to their customer value.

The conflict situations illustrated here can be mitigated or eliminated by creating fuzzy customer classes. The position of a customer in a two or more-dimensional data matrix relates to the customer value now consisting of different class membership fractions.

According to Fig. 6.16, a certain customer's loyalty as a linguistic variable can simultaneously be "positive" and "negative". For example, Bell belongs to the fuzzy set μpositive with a rate of 0.66 and to the set μnegative with 0.33, i.e., Bell's loyalty is not exclusively strong or weak, as it would be with sharp classes.

The linguistic variable μ with the vague terms "positive" and "negative" and the membership functions $\mu_{positive}$ and $\mu_{negative}$ results in the domain D(loyalty) being partitioned fuzzily. Analogously, the domain D(revenue) is partitioned by the terms "high" and "low". This allows for classes with gradual transitions (fuzzy classes) in the context model.

An object's membership in a class is the result of the aggregation across all terms defining that class. Class C1 is described by the terms "high" (for the linguistic variable revenue) and "positive" (for the linguistic variable loyalty). The aggregation, therefore, has to correspond to the conjunction of the individual membership values. For this, various operators have been developed in fuzzy set theory.

Classification queries in the *language fCQL* (fuzzy Classification Query Language) operate on the linguistic level with vague contexts. This has the advantage that users do not need to know sharp goal values or contexts, but only the column name of the value identifying the object and the table or view containing the attribute values. In order to

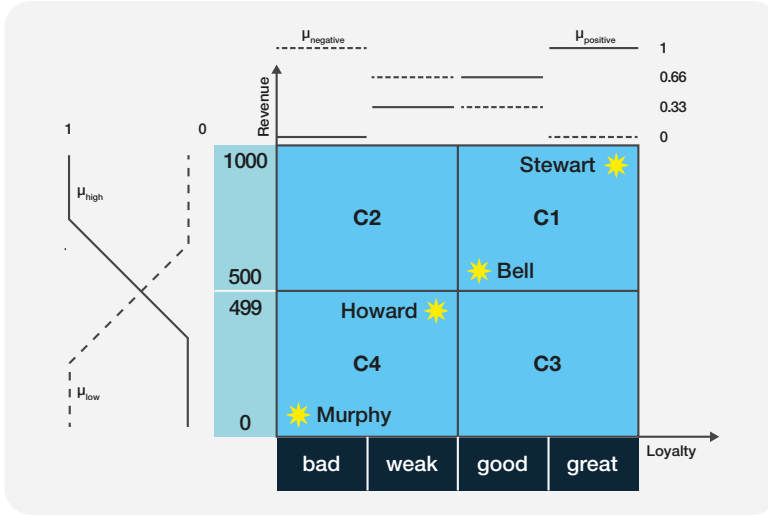


Fig. 6.16 Fuzzy partitioning of domains with membership functions

take a more detailed look at single classes, users can specify a class or state attributes with a verbal description of their intensity. Classification queries, therefore, work with verbal descriptions on attribute or class level:

```
CLASSIFY  Object
FROM      Table
WITH      Classification condition
```

The language fCQL is based on SQL, with a CLASSIFY clause instead of SELECT defining the projection list by the column name of the object to be classified. While the WHERE clause in SQL contains a selection condition, the WITH clause determines a classification condition. As an example of an fCQL query,

```
CLASSIFY  Customer
FROM      Customer table
```

provides a classification of all customers in the table. The query

```
CLASSIFY  Customer
FROM      Customer table
WITH      CLASS IS Increase revenue
```

specifically targets class C3. Bypassing the definition of a class, it is also possible to select a certain set of objects by using the linguistic descriptions of the equivalence classes. The following query is an example:

```
CLASSIFY  Customer
FROM      Customer table
WITH      Revenue IS small AND Loyalty IS strong
```

This query consists of the identifier of the object to be classified (Customer), the name of the base table (Customer table), the critical attribute names (Revenue and Loyalty), the term "small" of the linguistic variable Revenue, and the term "strong" of the linguistic variable Loyalty.

Based on the example and the explanations above, fuzzy databases can be characterized as follows:

▶ **Fuzzy database system** A fuzzy database management system (FDBMS) is a database system with the following properties:

- The data model is fuzzily rational, i.e., it accepts *imprecise, vague, and uncertain* attribute values.
- Dependencies between attributes are expressed with *fuzzy normal forms*.
- Relational calculus as well as relational algebra can be extended to *fuzzy relational calculus* and *fuzzy relational algebra* using fuzzy logic.
- Using a classification language enhanced with linguistic variables, *fuzzy queries can be formulated.*

Only a few computer scientists have been researching the field of fuzzy logic and relational database systems over the years (Sect. 6.9). Their works are mainly published and acknowledged in the field of fuzzy logic, not in the database field. It is to be hoped that both fields will grow closer and the leading experts on database technology will recognize the potential that lies in fuzzy databases and fuzzy query languages.

6.9 Further Reading

One major publication on distributed database systems is the book by Ceri and Pelagatti (1985), and another overview is offered by Özsu and Valduriez (1991). Dadam (1996) illustrates distributed databases and customer-server systems. The German-language work by Rahm (1994) covers aspects of distributed database systems as well as questions of parallel multicomputer systems. Fundamental works on distributed database systems are based on the extensions of "System R" by Williams et al. (1982) and "Ingres" by Stonebraker (1986). Of interest are also the works of Rothnie et al. (1980), since the

database system "SDD-1" covered there was the first prototype of a distributed system. There are several approaches to the integration of time into relational databases; for instance those by Clifford and Warren (1983), Gadia (1988), and Snodgrass (1987), to name only a few. Snodgrass (1994), in cooperation with research colleagues, enhanced the SQL language with temporal constructs; that language is known as TSQL2. Further research on temporal databases can be found in Etzion et al. (1998). Myrach (2005) covers temporal aspects in business information systems.

The standard for works on data warehouses is set by Inmon (2005), the source of the data warehouse definition used in this book. Kimball et al. (2008) also discussed the topic. Well-known German-language works are, e.g., Gluchowski et al. (2008) and Mucksch and Behme (2000). Jarke et al. (2000) convey the basics of data warehouses and propose research projects. Books on data mining were written by Berson and Smith (1997) and Witten and Frank (2005). The integration of databases in the World Wide Web is illustrated by several authors in a focus journal issue by Meier (2000).

Object-oriented approaches for extensions of relational databases are presented by Dittrich (1988), Lorie et al. (1985), Meier (1987), and Schek and Scholl (1986). Books on object-oriented databases come from Bertino and Martino (1993), Cattell (1994), Geppert (2002), Heuer (1997), Hughes (1991), and Kim (1990). Lausen and Vossen (1996) describe fundamental aspects of object-relational and object-oriented database languages. German works by Kemper and Eickler (2013), Lang and Lockemann (1995), and Saake et al. (1997) explain developments in relational and object-oriented database systems. Meier and Wüst (2003) wrote an introduction to object-oriented and object-relational databases for practical application. Stonebraker (1996) explains object-relational database systems. The development of the extension of the SQL standard is summarized in Türker (2003). The publications by Coad and Yourdon (1991) and Martin and Odell (1992) include building blocks for object-oriented database design. Stein (1994) provides a comparison of different object-oriented analysis methods in his book.

The rule-based language in deductive database systems is often called Datalog, taking inspiration from the word data and the well-known logic programming language Prolog. A major Prolog textbook is by Clocksin and Mellish (1994); a formal treatise on logical database programming can be found in Maier and Warren (1988). The German-language work by Cremers et al. (1994) comprehensively discusses deductive databases. The works of Gallaire et al. (1984) and Gardarin and Valduriez (1989) are largely dedicated to deductive databases.

The research area of fuzzy sets was founded by Lotfi A. Zadeh (Zadeh 1965), in part for the extension of classical logic, with the values "true" and "false", towards a fuzzy logic with any number of truth values. The results of this have been applied to data modeling and databases for some years now, see, for instance, Bordogna and Pasi (2000), Bosc and Kacprzyk (1995), Chen (1998), Petra (1996), and Pons et al. (2000). Using fuzzy logic, several model enhancements have been proposed for both the entity-relationship model and the relational model. For example, Chen (1992) in his dissertation developed the classic normal forms of the database theory into fuzzy ones by permitting

fuzziness in the functional dependencies; see also Shenoi et al. (1992). Other proposals regarding fuzzy data models can be found in Kerre and Chen (1995). Takahashi (1995) proposes a Fuzzy Query Language (FQL) based on relational calculus. The language FQUERY by Kacprzyk and Zadrozny (1995) uses fuzzy terms and has been implemented in Microsoft Access as a prototype. A slightly different approach is taken with fuzzy classification, as originally proposed by Schindler (1998). The fuzzy classification language fCQL (fuzzy Classification Query Language) was devised for this purpose and realized in a prototype (see Meier et al. 2008, 2005, Werro 2015).

References

Berson, A., Smith, S.: Data Warehousing, Data Mining and OLAP. McGraw-Hill, New York (1997)

Bertino E., Martino L.: Object-Oriented Database Systems. Addison Wesley, Wokingham (1993)

Bordogna G., Pasi G. (eds.): Recent Issues on Fuzzy Databases. Physica, Heidelberg (2000)

Bosc P., Kacprzyk J. (eds.): Fuzziness in Database Management Systems. Physica, Heidelberg (1995)

Cattell R. G.G.: Object Data Management—Object-Oriented and Extended Relational Database Systems. Addison Wesley, Reading (1994)

Ceri S., Pelagatti G.: Distributed Databases—Principles and Systems. McGraw-Hill, New York (1985)

Chen G.: Design of Fuzzy Relational Databases Based on Fuzzy Functional Dependenciey. PhD Thesis No. 84, Leuven, Belgium (1992)

Chen G.: Fuzzy Logic in Data Modeling—Semantics, Constraints, and Database Design. Kluwer, Boston (1998)

Clifford, J., Warren, D.S.: Formal semantics for time in databases. ACM Trans. Database Syst. **8**(2), 214–254 (1983)

Clocksin W.F., Mellish C.S.: Programming in Prolog. Springer, Berlin (1994)

Coad P., Yourdon E.: Object-Oriented Design. Yourdon Press, Englewood Cliffs (1991)

Cremers A.B. et al.: Deduktive Datenbanken—Eine Einführung aus der Sicht der logischen Programmierung. Vieweg, Braunschweig (1994)

Dadam P.: Verteilte Datenbanken und Client/Server-Systeme. Springer, Berlin (1996)

Dittrich K.R. (ed.): Advances in Object-Oriented Database Systems. Lecture Notes in Computer Science, Vol. 334. Springer, Berlin (1988)

Etzion O., Jajodia S., Sripada S. (eds.): Temporal Databases—Research and Practice. Lecture Notes in Computer Science. Springer, Berlin (1998)

Gadia, S.K.: A Homogeneous Relational Model and Query Languages for Temporal Databases. ACM Trans. Database Syst. **13**(4), 418–448 (1988)

Gallaire, H., et al.: Logic and databases: A deductive approach. ACM Comput. Surv. **16**(2), 153–185 (1984)

Gardarin G., Valduriez P.: Relational Databases and Knowledge Bases. Addison Wesley, Reading (1989)

Geppert A.: Objektrelationale und objektorientierte Datenbankkonzepte und -systeme. dpunkt, Heidelberg (2002)

Gluchowski P., Gabriel R., Chamoni P.: Management Support Systeme und Business Intelligence—Computergestützte Informationssysteme für Fach- und Führungskräfte. Springer, Berlin (2008)

Heuer A.: Objektorientierte Datenbanken. Addison Wesley, Reading (1997)
Hughes J.G.: Object-Oriented Databases. Prentice-Hall, New York (1991)
Inmon W.H.: Building the Data Warehouse. Wiley, Indianapolis (2005)
Jarke M., Lenzerini M., Vassiliou Y., Vassiliadis P.: Fundamentals of Data Warehouses. Springer, Berlin (2000)
Kacprzyk J., Zadrozny S.: FQUERY for access—Fuzzy querying for a windows-based DBMS. In: Bosc P., Kacprzyk J. (eds.) Fuzziness in Database Management Systems, pp. 415–433. Physica, Heidelberg (1995)
Kemper A., Eickler A.: Datenbanksysteme—Eine Einführung. Oldenbourg, München (2013)
Kim W.: Introduction to Object-Oriented Databases. The MIT Press, Cambridge (1990)
Kimball R., Ross M., Thorntwaite W., Mundy J., Becker B.: The Datawarehouse Lifecycle Toolkit. Wiley, Indianapolis (2008)
Lang S.M., Lockemann P. C.: Datenbankeinsatz. Springer, Berlin (1995)
Lausen G., Vossen G.: Objekt-orientierte Datenbanken: Modelle und Sprachen. Oldenbourg, Berlin (1996)
Lorie R.A., Kim W., McNabb D., Plouffe W., Meier A.: Supporting complex objects in a relational system for engineering databases. In: Kim W. et al. (eds.) Query Processing in Database Systems, pp. 145–155. Springer, Berlin (1985)
Maier D., Warren D.S.: Computing with Logic—Logic Programming with Prolog. Benjamin/Cummings, Menlo Park (1988)
Martin J., Odell J.J.: Object-Oriented Analysis and Design. Prentice-Hall, Englewood Cliffs (1992)
Meier A.: Erweiterung relationaler Datenbanksystems für technische Anwendungen. Informatik-Fachberichte, Vol. 135. Springer, Berlin (1987)
Meier A. (Hrsg.): WWW & Datenbanken. Praxis der Wirtschaftsinformatik, Vol. 37, No. 214. dpunkt, Heidelberg (2000)
Meier A., Wüst T.: Objektorientierte und objektrelationale Datenbanken—Ein Kompass für die Praxis. dpunkt, Heidelberg (2003)
Meier A., Werro N., Albrecht M., Sarakinos M.: Using a Fuzzy Classification Query Language for Customer Relationship Management. Proc. 31st International Conference on Very Large Databases (VLDB), Trondheim, Norway, pp. 1089–1096 (2005)
Meier A., Schindler G., Werro N.: Fuzzy classification on relational databases (Chapter XXIII). In: Galindo J. (ed.) Handbook of Research on Fuzzy Information Processing in Databases. Vol. II, pp. 586–614. IGI Global, Pennsylvania (2008)
Mucksch, H., Behme, W. (eds.): Das Data-Warehouse-Konzept—Architektur. Datenmodelle, Anwendungen. Gabler, Wiesbaden (2000)
Myrach, T.: Temporale Datenbanken in betrieblichen Informationssystemen—Prinzipien, Konzepte, Umsetzung. Teubner, Wiesbaden (2005)
Özsu M.T., Valduriez P.: Principles of Distributed Database Systems. Prentice Hall, Englewood Cliffs (1991)
Petra F.E.: Fuzzy Databases—Principles and Applications. Kluwer, Boston (1996)
Pons O., Vila M.A., Kacprzyk J. (eds.): Knowledge Management in Fuzzy Databases. Physica, Heidelberg (2000)
Rahm E.: Mehrrechner-Datenbanksysteme. Addison Wesley, Bonn (1994)
Rothnie, J.B., et al.: Introduction to a System for Distributed Databases. ACM Trans. Database Syst. **5**(1), 1–17 (1980)
Schek, H.-J., Scholl, M.H.: The Relational Model with Relation-Valued Attributes. Inf. Syst. **11**(2), 137–147 (1986)
Schindler G.: Fuzzy Datenanalyse durch kontextbasierte Datenbankabfragen. Deutscher Universitäts-Verlag, Wiesbaden (1998)

Shenoi, S., Melton, A., Fan, L.T.: Functional Dependencies and Normal Forms in the Fuzzy Relational Database Model. Inf. Sci. **60**, 1–28 (1992)
Snodgrass, R.T.: The Temporal Query Language TQuel. ACM Trans. Database Syst. **12**(2), 247–298 (1987)
Snodgrass, R.T., et al.: A TSQL2 Tutorial. SIGMOD-Record **23**(3), 27–33 (1994)
Stein, W.: Objektorientierte Analysemethoden—Vergleich, Bewertung, Auswahl. Bibliographisches Institut, Mannheim (1994)
Stonebraker M.: The Ingres Papers. Addison Wesley, Reading (1986)
Stonebraker M.: Object-Relational DBMS's—The Next Great Wave. Morgan Kaufmann, San Francisco (1996)
Takahashi Y.: A Fuzzy Query Language for Relational Databases. In: Bosc P., Kacprzyk J. (eds.) Fuzzyness in Database Management Systems, pp. 365–384. Physica, Heidelberg (1995)
Türker C.: SQL:1999 & SQL:2003—Objektrelationales SQL, SQLJ & SQL/XML. dpunkt, Heidelberg (2003)
Werro N.: Fuzzy Classification of Online Customers. Springer, Heidelberg (2015)
Williams R. et al.: R*: An Overview of the Architecture. In: Scheuermann P. (ed.) Improving Database Usability and Responsiveness, pp. 1–27. Academic, New York (1982)
Witten I. H., Frank E.: Data Mining. Morgan Kaufmann, Amsterdam (2005)
Zadeh L. A.: Fuzzy Sets. Information and Control, No. 8, pp. 338–353 (1965)

NoSQL Databases

7

7.1 Development of Nonrelational Technologies

The term NoSQL was first used in 1998 for a database that (although relational) did not have an SQL interface. It became of growing importance during the 2000s, especially with the rapid expansion of the internet. The growing popularity of global web services saw an increase in the use of *web-scale databases*, since there was a need for data management systems that could handle the enormous amounts of data (sometimes in the petabyte range and up) generated by web services.

Relational/SQL database systems are much more than mere data storage systems. They provide a large degree of processing logic:

- Powerful declarative language constructs
- Schemas and metadata
- Consistency assurance
- Referential integrity and triggers
- Recovery and logging
- Multi-user operation and synchronization
- Users, roles, and security
- Indexing

These SQL functionalities offer numerous benefits regarding data consistency and security. This goes to show that SQL databases are mainly designed for integrity and transaction protection, as required in banking applications or insurance software, among others. However, since data integrity control requires much work and processing power, relational databases quickly reach their limits with large amounts of data. The powerfulness of the database management system is disadvantageous for efficiency and performance, as well as for flexibility in data processing.

A. Meier and M. Kaufmann, *SQL & NoSQL Databases,*
https://doi.org/10.1007/978-3-658-24549-8_7

In practical use, consistency-oriented processing components often impede the efficient processing of huge amounts of data, especially in use cases where the focus is on performance rather than consistency, such as social media. That is why the open source and web development communities soon began to push the development of massive distributed database systems that can fulfill these new demands.

▶ **NoSQL database** NoSQL databases have the following properties (Sect. 1.4.3):

- The database model is not relational.
- The focus is on distributed and horizontal scalability.
- There are weak or no schema restrictions.
- Data replication is easy.
- Easy access is provided via an API.
- The consistency model is not ACID (instead, e.g., BASE, Sect. 4.2.1).

Although NoSQL primarily reads as databases that provide no SQL access, the acronym is commonly defined as "not only SQL". Different database models are suitable for different purposes, and the use of various database types within one application can be beneficial if each is used according to its strengths. This concept is called *polyglot persistence* and allows for both SQL and NoSQL technologies to be deployed within one application.

Core NoSQL technologies are:

- Key-value stores (Sect. 7.2)
- Column family databases (Sect. 7.3)
- Document stores (Sect. 7.4)
- Graph databases (Sect. 7.6).

These four database models, also called *core NoSQL models*, are discussed in this chapter. Other types of NoSQL databases fall in the category of Soft NoSQL, e.g., object databases, grid databases, and the family of XML databases (Sect. 7.5).

7.2 Key-Value Stores

The simplest way of storing data is assigning a value to a variable or a key. At the hardware level, CPUs work with registers based on this model; programming languages use the concept in associative arrays. Accordingly, the simplest database model possible is data storage that stores *a data object as a value for another data object as key*.

In *key-value stores*, a specific value can be stored for any key with a simple command, e.g., SET. Below is an example in which data for users of a website is stored: first name, last name, e-mail, and encrypted password. For instance, the value John is stored for the key User:U17547:firstname.

```
SET User:U17547:firstname John
SET User:U17547:lastname Doe
SET User:U17547:email john.doe@blue_planet.net
SET User:U17547:pwhash D75872C818DC63BC1D87EA12
SET User:U17548:firstname Jane
SET User:U17548:lastname Doherty
…
```

Data objects can be retrieved with a simple query using the key:

```
GET User:U17547:email
> john.doe@blue_planet.net
```

The key space can only be structured with special characters such as colons or slashes. This allows for the definition of a namespace that can represent a rudimentary data structure. Apart from that, key-value stores do not support any kind of structure, neither nesting nor references. Key-value stores are schema-less, i.e., data objects can be stored at any time and in arbitrary formats, without the need for any metadata objects such as tables or columns to be defined beforehand. Going without a schema or referential integrity makes key-value stores performant for queries, easy to partition, and flexible regarding the types of data to be stored.

▶ **Key-value store** A database is a key-value store if it has the following properties:

- There is a set of identifying data objects, the *keys*.
- For each key, there is exactly one associated descriptive data object, the *value* for that key.
- Specifying a key allows querying the associated value in the database.

Key-value stores have seen a large increase in popularity as part of the NoSQL trend, since they are scalable for huge amounts of data. As referential integrity is not checked in key-value stores, it is possible to write and read extensive amounts of data efficiently. Processing speed can be enhanced even further if the key-value pairs are buffered in the main memory of the database. Such setups are called *in-memory databases*. They employ technologies that allow to cache values in the main memory while constantly validating them against the long-term persistent data in the background memory.

There is almost no limit to increasing a key-value store's scalability with fragmentation or *sharding* of the data content. Partitioning is rather easy in key-value stores, due to the simple model. Individual computers within the cluster, called *shards*, take on only a part of the keyspace. This allows for the distribution of the database onto a large number of individual machines. The keys are usually distributed according to the principles of consistent hashing (Sect. 5.2.3).

Figure 7.1 shows a distributed architecture for a key-value store: A numerical value (hash) is generated from a key; using the module operator, this value can now be positioned on a defined number of address spaces (hash slots) in order to determine on which shard within the distributed architecture the value for the key will be stored. The distributed database can also be copied to additional computers and updated there to improve partition tolerance, a process called replication. The original data content in the master cluster is synchronized with multiple replicated data sets, the slave clusters.

Figure 7.1 shows an example of a possible massively distributed high-performance architecture for a key-value store. The master cluster contains three computers (shards A, B, and C). The data is kept directly in the main memory (RAM) to reduce response times. The data content is replicated to a slave cluster for permanent storage on a hard drive. Another slave cluster further increases performance by providing another replicated computer cluster for complex queries and analyses.

Apart from the efficient sharding of large amounts of data, another advantage of key-value stores is the flexibility of the data schema. In a relational database, a pre-existing schema in the shape of a relation with attributes (CREATE TABLE) is necessary for any record to be stored. If there is none, a schema definition must be executed before saving the data. For database tables with large numbers of records or for the insertion of heterogeneous data, this is often a lot of work. Key-value stores are basically schema-free and therefore highly flexible regarding the type of data to be stored. It is not necessary to

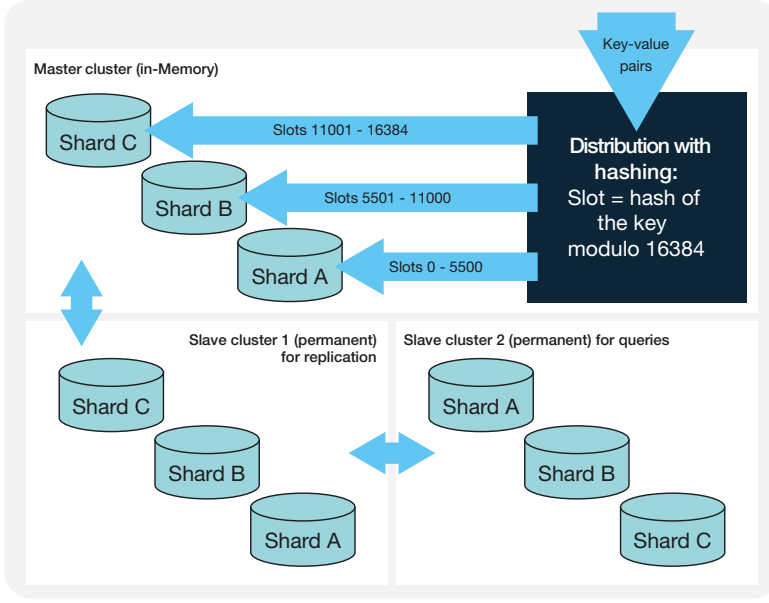


Fig. 7.1 Massively distributed key-value store with sharding and hash-based key distribution

specify a table with columns and data types, rather, the data can simply be stored under an arbitrary key. On the other hand, the lack of a database schema often causes a clutter in data management.

7.3 Column-Family Stores

Even though key-value stores are able to process large amounts of data performantly, their structure is still quite rudimentary. Often, the data matrix needs to be structured with a schema. *Column-family stores* enhance the key-value concept accordingly by providing additional structure.

In practical use, it has shown to be more efficient for optimizing read operations to store the data in relational tables not per row, but per column. This is because all columns in one row are rarely needed at once, but there are *groups of columns that are often read together*. Therefore, in order to optimize access, it is useful to structure the data in such groups of columns—column families—as storage units. Column-family stores, which are named after this method, follow this model; they store data not in relational tables, but in enhanced and structured multidimensional key spaces.

Google presented its Bigtable database model for the distributed storage of structured data in 2008, significantly influencing the development of column-family stores.

▶ **Bigtable** In the Bigtable model, a *table* is a sparse, distributed, multidimensional, sorted map. It has the following properties:

- The data structure is a map which assigns elements from a domain to elements in a co-domain.
- The mapping function is sorted, i.e., there is an order relation for the keys addressing the target elements.
- The addressing is multidimensional, i.e., the function has more than one parameter.
- The data is distributed by the map, i.e., it can be stored on different computers in different places.
- The map is sparse, since there are possibly many keys without data entry.

In Bigtable, a table has three dimensions: It maps an entry of the database for one *row* and one *column* at a certain *time* as a string:

```
(row:string, column:string, time:int64) ←string
```

Tables in column-family stores are multistage aggregated structures. The first key, the row key, is an addressing of a database object, as in a key-value store. Within this key, however, there is another structure, dividing the row into several columns, which are also addressed with keys. Entries in the table are additionally versioned with a time stamp.

The storage unit addressed with a certain combination of row key, column key, and time stamp is called a *cell*.

Columns in a table are grouped into column families. These are the unit for access control, i.e., for granting reading and writing permissions to users and applications. Additionally, the unit of the column family is used in assigning main memory and hard drive space. Column families are *the only fixed schema rules* of the table, which is why they need to be created explicitly by changing the schema of the table. Unlike in relational databases, various row keys can be used within one column family to store data. The column family, therefore, serves as a *rudimentary schema* with a reduced amount of metadata.

Data within a column family is of the same type, since it is assumed it will be read together. This is also why the database always stores the data of one column family in one row of the table on the same computer. This mechanism reduces the time needed for combined reading access within the column family. Therefore, the database management system sorts column families into *locality groups,* which define on which computer and in which format the data is stored. The data of one locality group is physically stored on the same computer. Additionally, it is possible to set certain parameters for locality groups, for instance to keep a specific locality group in the main memory; making it possible to read the data quickly without the need to access the hard drive.

Figure 7.2 summarizes how data is stored in the Bigtable model described above: A data cell is addressed with row key and column key. In the given example, there is one row key per user. The content is additionally historicized with a time stamp. Several columns are grouped into column families: The columns Mail, Name, and Phone form the

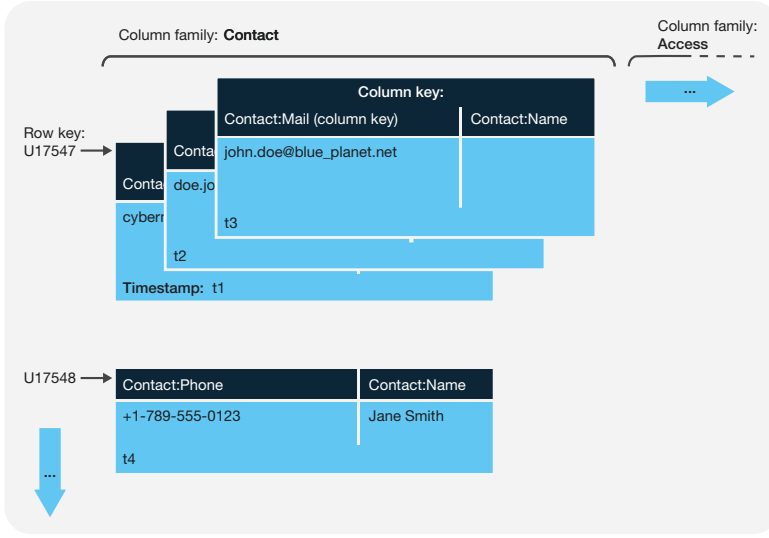


Fig. 7.2 Storing data in the Bigtable model

column family Contact. Access data, such as user names and passwords could be stored in the column family Access. The columns in a column family are *sparse*. In the example in Fig. 7.2, the row U17547 contains a value for the column Contact:Mail, but not for the column Contact:Phone. If there is no entry, this information will not be stored in the row.

▶ **Column-family store** Databases using a data model similar to the Bigtable model are called column-family stores. They can be defined as NoSQL databases with the following properties:

- The data is stored in multidimensional tables.
- Data objects are addressed with row keys.
- Object properties are addressed with column keys.
- Columns of the tables are grouped into column families.
- A table's schema only refers to the column families; within one column family, arbitrary column keys can be used.
- In distributed, fragmented architectures, the data of a column family is preferably physically stored at one place (co-location) in order to optimize response times.

The advantages of column-family stores are their high scalability and availability due to their massive distribution, just as with key-value stores. Additionally, they provide a useful structure with a schema offering access control and localization of distributed data on the column family level; at the same time they provide enough flexibility within the column family by making it possible to use arbitrary column keys.

7.4 Document Stores

A third variety of NoSQL databases, *document stores*, combines the absence of a schema with the possibility of structuring the stored data. Unlike what is implied by the name, document stores do not store arbitrary documents such as web, video, or audio data, but structured data in records which are called *documents*.

The usual document stores were developed specifically for the use in web services. They can, therefore, easily be integrated with web technologies such as JavaScript and HTTP[1]. Additionally, they are readily horizontally scalable by combination of several computers into an integrated system which distributes the data volume by sharding. The focus is mostly on processing large amounts of heterogeneous data, while for most web data, for instance from social media, search engines, or news portals, the constant consistency of data does not need to be ensured. Security sensitive web services such as

[1]HyperText Transfer Protocol.

online banking, which heavily rely on schema restrictions and guaranteed consistency, are an exception.

Document stores are completely schema-free, i.e., there is no need to define a schema before inserting data structures. The schematic responsibility is, therefore, transferred to the user or the processing application. The disadvantage arising from not having a schema is the missing referential integrity and normalization. However, the absence of a schema allows for extreme flexibility in storing a wide range of data, which is what Variety in the Vs of Big Data (see Sect. 1.3) refers to. This also facilitates fragmentation of the data.

On the first level, document stores are a kind of key-value stores. For every key (document ID), a record can be stored as value. These records are called *documents*. On the second level, these documents have their *own internal structure*. The term document is not entirely appropriate, since they are explicitly not multimedia or other unstructured data. A document in the context of a document store is a file with structured data, for instance in JSON[2] format. The structure is a list of attribute-value pairs. All attribute values in this data structure can recursively contain lists of attribute-value pairs themselves. The documents are not connected to each other, but contain a closed collection of data.

Figure 7.3 shows a sample document store D_USERS that stores data on the users of a website. For every user key with the attribute _id, an object containing all user information, such as user name, first name, last name, and gender, is stored. The visitHistory attribute holds a nested attribute value as an associative array, which again contains key-value pairs. This nested structure lists the date of the last visit to the website as the associated value.

Apart from the standard attribute _id, the document contains a field _rev (revision), which indexes the version of the document. One possibility of resolving concurring queries is *multiversion concurrency control*. The database makes sure that every query receives the revision of a document with the largest number of changes. As this cannot ensure full transactional security, it is called *eventual consistency*. The consistency of the data is only reached after some time. This significantly speeds up data processing at the expense of transactional security.

▶ **Document store** To summarize, a document store is a database management system with the following properties:

- It is a key-value store.
- The data objects stored as values for keys are called documents; the keys are used for identification.

[2]JavaScript Object Notation.

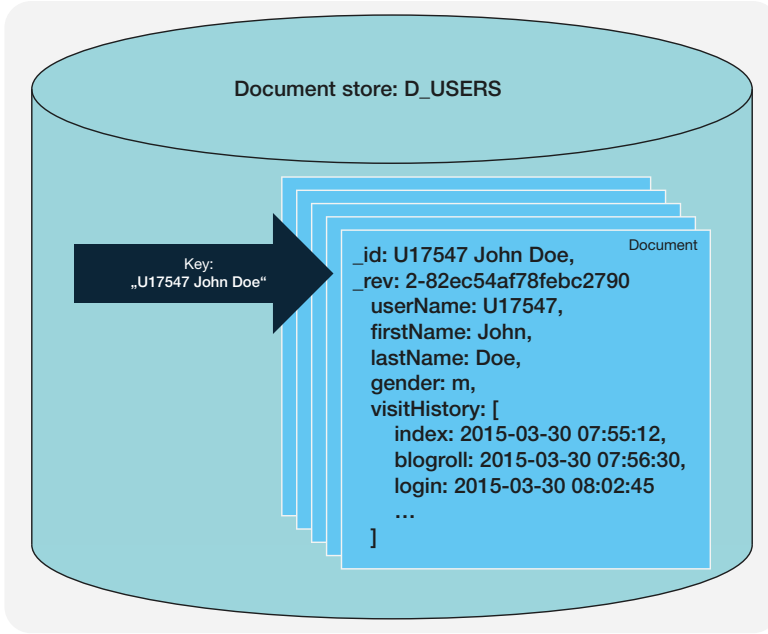


Fig. 7.3 Example of a document store

- The documents contain data structures in the form of recursively nested attribute-value pairs without referential integrity.
- These data structures are schema-free, i.e., arbitrary attributes can be used in every document without defining a schema first.

Queries on a document store can be parallelized and, therefore, sped up with the *MapReduce procedure* (Sect. 5.4). Such processes are two-phased, where Map corresponds to grouping (group by) and Reduce corresponds to aggregation (count, sum, etc.) in SQL.

During the first phase, a map function which carries out a predefined process for every document is executed, building and returning a *map*. Such a map is an associative array with one or several key-value pairs per document. The map phase can be calculated per document independently from the rest of the data content, thereby always allowing for parallel processing without dependencies if the database is distributed among different computers.

In the optional reduce phase, a function is executed to *reduce* the data, returning one row per key in the index from the map function and aggregating the corresponding values. The following example demonstrates how Map/Reduce can be used to calculate the number of users, grouped by gender, in the database from Fig. 7.3.

Because of the absence of a schema, as part of the map function a check is executed for every document to find out if the attribute userName exists. If that is the case, the `emit` function returns a key-value pair, with the key being the user's gender, the value the number 1. The reduce function then receives two different keys, `m` and `f`, in the `keys` array, and for every document per user of the respective gender a number 1, as values in the `values` array. The reduce function returns the sum of the ones, grouped by key, which equals the respective number.

```
// map
function(doc){
 if(doc.userName) {
 emit(doc.gender, 1)
 }
}
// reduce
function(keys, values) {
return sum(values)
}
// > key value
// >"f" 456 // > "m" 567
```

The results of Map/Reduce processes, called views, should be precalculated and indexed as permanent views using design documents for an optimal performance. Key-value pairs in document stores are stored in B-trees (Sect. 5.2.1). This allows quick access to individual key values. The reduce function uses a B-tree structure by storing aggregates in balanced trees, with only a few detail values stored in the leaves. Updating aggregates, therefore, only requires changes to the respective leaf and the (few) nodes with subtotals down to the root.

7.5 XML Databases

XML (eXtensible Markup Language) was developed by the World Wide Web Consortium (W3C). The content of hypertext documents is marked by tags, just as in HTML. An XML document is self-describing, since it contains not only the actual data, but also information on the data structure.

```
<address>
<street> W Broad Street </street>
<number> 333 </number>
<ZIP code> 43215 </ZIP code>
<city> Columbus </city>
</address>
```

The basic building blocks of XML documents are called elements. They consist of a start tag (in angle brackets <name>) and an end tag (in angle brackets with slash </name>) with the content of the element in-between. The identifier of the start and the end tag must match.

The tags provide information on the meaning of the specific values and, therefore, make statements about the data semantics. Elements in XML documents can be nested arbitrarily. It is best to use a graph to visualize such hierarchically structured documents, as shown in the example in Fig. 7.4.

As mentioned above, XML documents also implicitly include information about the structure of the document. Since it is important for many applications to know the structure of the XML documents, explicit representations (DTD = Document Type Definition or XML schema) have been proposed by W3C. An explicit schema shows which tags occur in the XML document and how they are arranged. This allows for, e.g., localizing and repairing errors in XML documents. The XML schema is illustrated here as it has undeniable advantages for use in database systems.

An XML schema and a relational database schema are related as follows: Usually, relational database schemas can be characterized by three degrees of element nesting, i.e., the name of the database, the relation names, and the attribute names. This makes it

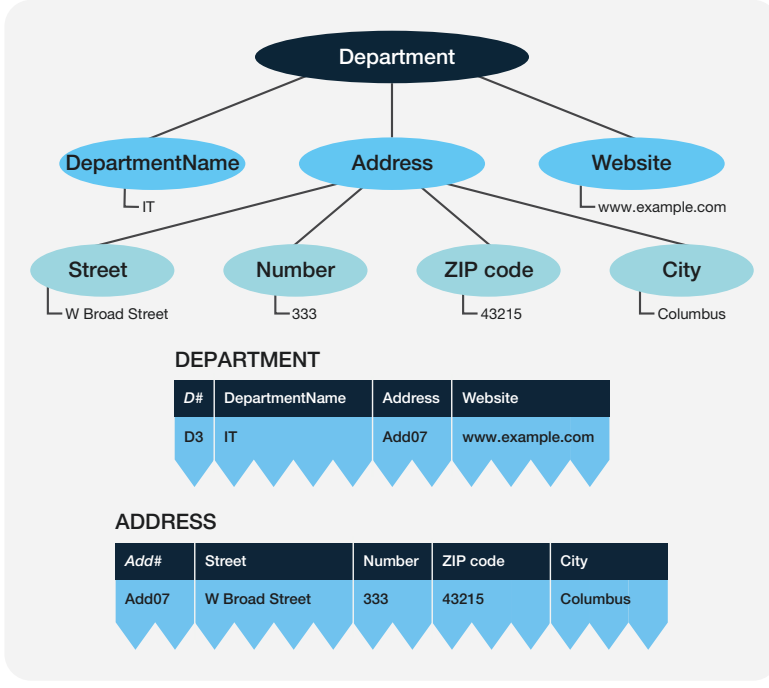


Fig. 7.4 Illustration of an XML document represented by tables

possible to match a relational database schema to a section of an XML schema and vice versa.

Figure 7.4 shows the association between an XML document and a relational database schema. The section of the XML document gives the relation names DEPARTMENT and ADDRESS, each with their respective attribute names and the actual data values. The use of keys and foreign keys is also possible in an XML schema, as explained below.

The basic concept of XML schemas is to define data types, and match names and data types using declarations. This allows for the creation of completely arbitrary XML documents. Additionally, it is possible to describe integrity rules for the correctness of XML documents.

There is a large number of standard data types, such as string, Boolean, integer, date, time, etc.; but apart from that, user-defined data types can also be introduced. Specific properties of data types can be declared with facets. This allows for the properties of a data type to be specified, for instance the restriction of values by an upper or lower limit, length restrictions, or lists of permitted values:

```
<xs:simpleType name=«city»>
<xs:restriction base=«xs:string»>
<xs:length value=«20»/>
</xs:restriction>
</xs:simpleType>
```

For cities, a simple data type based on the predefined data type string is proposed. Additionally, the city names cannot consist of more than 20 characters.

Several XML editors have been developed that allow for the graphical representation of an XML document or schema. These editors can be used for both the declaration of structural properties and the input of data content. By showing or hiding individual substructures, XML documents and schemas can be arranged neatly.

It is desirable to be able to analyze XML documents or XML databases. Unlike relational query languages, selection conditions are not only linked to values (value selection), but also to element structures (structure selection). Other basic operations of an XML query include the extraction of subelements of an XML document and the modification of selected subelements. Furthermore, individual elements from different source structures can be combined to form new element structures. Last but not least, a suitable query language needs to be able to work with hyperlinks; path expressions are vital for that.

XQuery, influenced by SQL, various XML languages (e.g., XPath as navigation language for XML documents) and object-oriented query languages, was proposed by the W3C. XQuery is an enhancement of XPath, offering not only the option to query data in XML documents, but also to form new XML structures. The basic elements of XQuery are *FOR-LET-WHERE-RETURN expressions*: FOR and LET bind one or more variables

to the results of a query of expressions. WHERE clauses can be used to further restrict the result set, just as in SQL. The result of a query is shown with RETURN.

A simple example to give an outline of the principles of XQuery: The XML document "Department" (Fig. 7.4) is queried for the street names of the individual departments:

```
<streetNames>
{FOR $Department IN //department RETURN
$Department/address/street }
</streetNames>
```

The query above binds the variable $Department to the <Department> nodes during processing. For each of these bindings the RETURN expression evaluates the address and returns the street. The query in XQuery produced the following result:

```
<streetNames>
<street> W Broad Street </street>
<street>..... </street>
<street>..... </street>
</streetNames>
```

In XQuery, variables are marked with the $ sign added to their names, in order to distinguish them from the names of elements. Unlike in some other programming languages, variables cannot have values assigned to them in XQuery; rather, it is necessary to analyze expressions and bind the result to the variables. This variable binding is done in XQuery with the FOR and LET expressions.

In the query example above, no LET expression is specified. Using the WHERE clause, the result set could be reduced further. The RETURN clause is executed for every FOR loop, but does not necessarily yield a result. The individual results, however, are listed and form the result of the FOR-LET-WHERE-RETURN expression.

XQuery is a *powerful query language for hyper documents* and is offered for XML databases as well as some postrelational database systems. In order for relational database systems to store XML documents, some enhancements in the storage component need to be applied.

Many relational database systems are nowadays equipped with XML column data types and, therefore, the possibility to directly handle XML. This allows for data to be stored in structured XML columns and for elements of the XML tree to be queried and modified directly with XQuery or XPath. Around the turn of the millennium, XML documents for data storage and data communication experienced a boom and were used for countless purposes, especially web services. As part of this trend, several database systems that can directly process data in the form of XML documents were developed. Particularly in the field of open source, support for XQuery in native XML databases is far stronger than in relational databases.

▶ **Native XML database** A native XML database is a database that has the following properties:

- The data is stored in documents; the database is, therefore, a document store (Sect. 7.4).
- The structured data in the documents is compatible with the XML standard.
- XML technologies such as XPath, XQuery, and XSL/T can be used for querying and manipulating data.

Native XML databases store data strictly hierarchically in a tree structure. They are especially suitable if hierarchical data needs to be stored in a standardized format, for instance for web services in service-oriented architectures (SOA). A significant advantage is the simplified data import into the database; some database systems even support drag & drop of XML files. Figure 7.5 shows a schematic illustration of a native XML database. It facilitates reading and writing access to data in a collection of XML documents for users and applications.

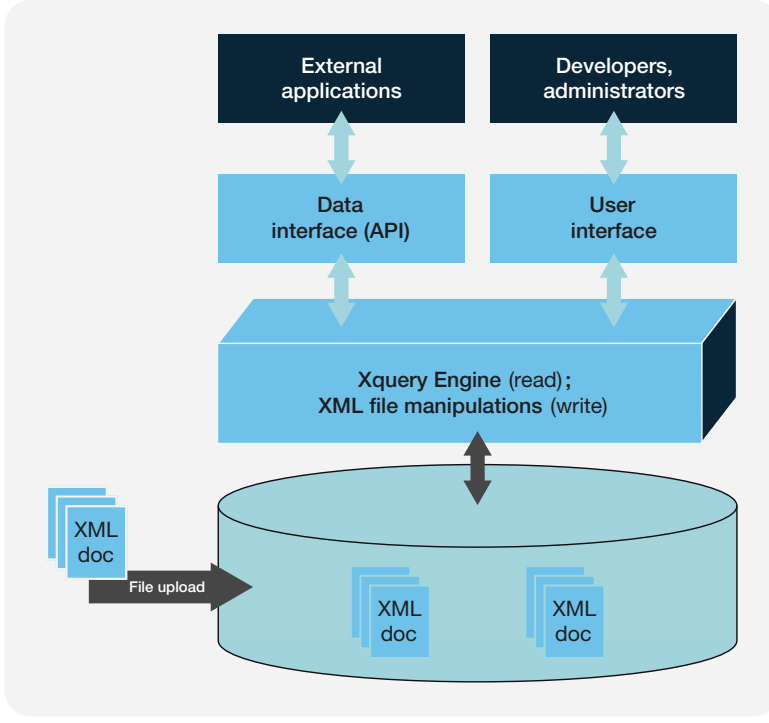


Fig. 7.5 Schema of a native XML database

An XML database cannot cross-reference like nodes. This can be problematic especially with multidimensionally-linked data. An XML database, therefore, is best suited for data that can be represented in a tree structure as a series of nested generalizations or aggregations.

7.6 Graph Databases

The fourth and final type of core NoSQL databases differs significantly from the data models presented up to this point, i.e., the key-value stores, column-family stores, and document stores. Those three data models forgo database schemas and referential integrity for the sake of easier fragmentation (sharding). Graph databases, however, have a structuring schema: that of the property graph presented in Sect. 1.4.1. In a graph database, data is stored as nodes and edges, which belong to a node type or edge type, respectively, and contain data in the form of attribute-value pairs. Unlike in relational databases, their schema is implicit, i.e., data objects belonging to a not-yet existing node or edge type can be inserted directly into the database without defining the type first. The DBMS implicitly follows the changes in the schema based on the information available and thereby creates the respective type.

As an example, Fig. 7.6 illustrates the graph database G_USERS, which represents information on a web portal with users, webpages, and the relationships between them. As explained in Sect. 1.4.1, the database has a schema with node and edge types. There are two node types, USER and WEBPAGE, and three edge types, FOLLOWS, VISITED, and CREATED_BY. The USER node type has the attributes userName, firstName, and lastName; the node type WEBPAGE has only the attribute Name; and the edge type VISITED has one attribute as well, date with values from the date domain. Therefore, it is a property graph.

This graph database stores a similar type of data as the D_USERS document database in Fig. 7.3; for instance, it also represents users with username, first name, last name, and the webpages visited with date. There is an important difference though: *The relationships between data objects are explicitly present as edges,* and referential integrity is ensured by the DBMS.

Graph database A *graph database* is a database management system with the following properties:

- The data and/or the schema are shown as *graphs* (Sect. 2.4) or graph-like structures, which generalize the concept of graphs (e.g., hypergraphs)
- Data manipulations are expressed as graph transformations, or operations which directly address typical properties of graphs (e.g., paths, adjacency, subgraphs, connections, etc.).

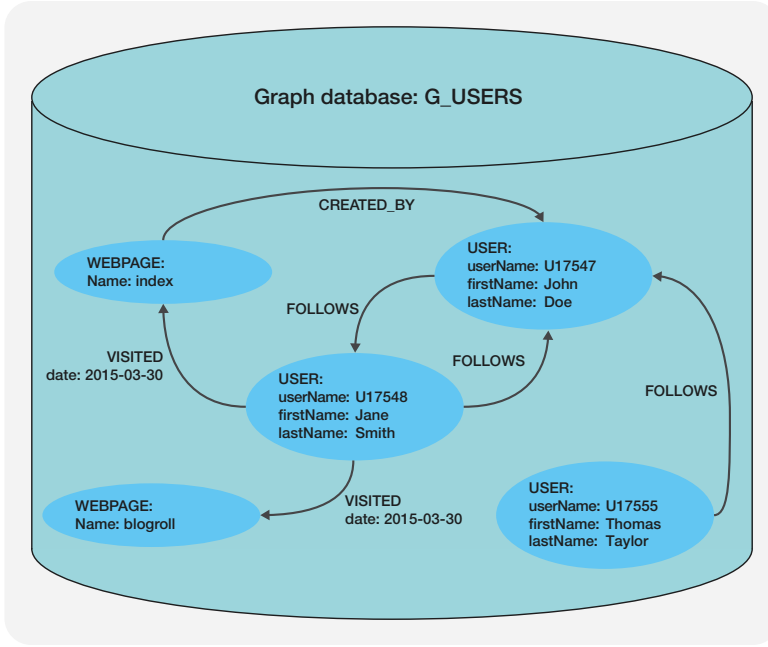


Fig. 7.6 Example of a graph database with user data of a website

- The database supports the checking of integrity constraints to ensure data consistency. The definition of consistency is directly related to graph structures (e.g., node and edge types, attribute domains, and referential integrity of the edges).

Graph databases are used when data is organized in networks. In these cases, it is not the individual record that matters, but the connection of all records with each other, for instance in social media, but also in the analysis of infrastructure networks (e.g., water network or electricity grid), in internet routing, or in the analysis of links between websites. The advantage of the graph database is the *index-free adjacency* property: For every node, the database system can find the direct neighbor, without having to consider all edges, as would be the case in relational databases using a relationship table. Therefore, the effort for querying the relationships with a node is constant, independent of the volume of the data. In relational databases, the effort for determining referenced tuples increases with the number of tuples, even if indexes are used.

Just as relational databases, graph databases need indexes to ensure a quick and direct access to individual nodes and edges via their properties. As illustrated in Sect. 5.2.1, balanced trees (B-trees) are generated for indexing. A tree is a special graph that does not contain any cycles; therefore every tree can be represented as a graph. This is interesting for graph data bases, because it means that the index of a graph can be a subgraph of the same graph. The graph contains its own indexes.

The fragmentation (Sect. 6.2) of graphs is somewhat more complicated. One reason why the other types of core NoSQL databases do not ensure relationships between records is that records can be stored on different computers with fragmentation (sharding) without further consideration, since there are no dependencies between them. The opposite is true for graph databases. Relationships between records are the central element of the database. Therefore, when fragmenting a graph database, the connections between records have to be taken into account, which often demands domain-specific knowledge. There is, however, no efficient method to optimally divide a graph into subgraphs. The existing algorithms are NP-complete, which means the computational expense is exponential. As a heuristic, clustering algorithms can determine highly interconnected partial graphs as partitions. Today's graph databases, however, do not yet support sharding.

7.7 Further Reading

Edlich et al. (2011) present the history of NoSQL databases. The definition of the term NoSQL database given there is the basis for the one used in this chapter. The term Polyglot Persistence is explained in Sadalage and Fowler's (2013) book.

Clustering and replication of a column-value database is described in the Cluster Tutorial on Redis' (2015) website. The data model for column-family stores is illustrated by Sadalage and Fowler (2013).

Google's Bigtable data structure was originally published by Chang et al. (2008).

As an example of a document store, Anderson et al. (2010) give an overview of CouchDB; including elaborations on data structure, views, design documents, and consistency in CouchDB, which are used prototypically and generally in this chapter. For the derivation of a grouped aggregate using MapReduce, it was thankfully possible to draw on a blog entry by Toby Ho (2009).

XML databases are discussed by Sadalage and Fowler (2013), whose definition was broadly adopted for this chapter. Additionally, XML databases are described by McCreary and Kelly (2014). XQuery, detailed instructions on the use of XML in relational databases, and guidelines for a native XML database can be found in the work of Fawcett et al. (2012).

A good reference for general work with graph-based data is available by Charu und Haixun (2010). Angles and Gutierrez (2008) give an overview over current graph database models; their work also provides the definition of graph databases used here. The work of Edlich et al. (2011) presents the graph model in detail and is also the source of the related information on use cases, indexing, and partitioning. Information on the index-free adjacency property and the complexity of algorithms for the sharding of graph databases are taken from Montag (2013).

References

Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide. O'Reilly. http://guide.couchdb.org/editions/1/en/index.html (2010). Accessed 23 March 2015

Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1), 1–39 (2008)

Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows M., Chandra, D., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. **26**(2), 1–26 (2008)

Charu, A., Haixun, W.: Managing and Mining Graph Data, Vol. 40. Springer, Dordrecht (2010)

Edlich, S., Friedland, A., Hampe, J., Brauer B., Brückner M.: NoSQL—Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken. Hanser, München (2011)

Fawcett, J., Quin, L. R. E, Ayers, D.: Beginning XML. Wiley, Indianapolis (2012)

Ho, T.: Taking an Average in CouchDB. http://tobyho.com/2009/10/07/taking-an-average-in-couchdb/ (2015). Accessed 23 March 2015

McCreary, D., Kelly, A.: Making Sense of NoSQL—A Guide for Managers and the Rest of Us. Manning, Shelter Island (2014)

Montag, D.: Understanding Neo4j Scalability. White Paper, netechnology. http://info.neo4j.com/rs/neotechnology/images/Understanding%20Neo4j%20Scalability(2).pdf (2013). Accessed 25 March 2015

Redis: Redis Cluster Tutorial. http://redis.io/topics/cluster-tutorial (2015). Accessed 2 March 2015

Sadalage, P.J., Fowler, M.: NoSQL Distilled—A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, Upper Saddle River (2013)

Glossary

ACID ACID is an acronym for atomicity, consistency, isolation, and durability. This abbreviation expresses that all transactions in a database lead from a consistent state to a new consistent state of the database.

Aggregation Aggregation describes the joining of entity sets into a whole. Aggregation structures can be network-like or hierarchical (item list).

Anomaly Anomalies are records that diverge from reality and can be created during insert, change, or delete operations in a database.

Association The association of one entity set to another is the meaning of the relationship in that direction. Associations can be weighted with an association type defining the cardinality of the relationship direction.

BASE BASE is an acronym for Basically Available, Soft state, Eventual consistency, meaning that a consistent state in a distributed database is reached eventually, with a delay.

Big Data The term Big Data describes data records that meet at least the three characteristic Vs: Volume—massive amounts of data ranging from terabytes to zettabytes; Variety—a multitude of structured, semi-structured, and unstructured data types, and Velocity—high-speed data stream processing.

Business Intelligence Business intelligence (BI) is a company-wide strategy for the analysis and the reporting of relevant business data.

CAP theorem The CAP (consistency, availability, partition tolerance) theorem states that in any massive distributed data management system, only two of the three properties consistency, availability, and partition tolerance can be ensured.

Column-family store Column stores or column-family stores are NoSQL databases in which the data is organized in columns or sets of columns.

Concurrency control Synchronization is the coordination of simultaneous accesses to a database in multi-user operations. Pessimistic concurrency control prevents conflicts between concurrent transactions from the start, while optimistic concurrency control resets conflicting transactions after completion.

Cursor management Cursor management enables the record-by-record processing of a set of data records with the help of a cursor.

A. Meier and M. Kaufmann, *SQL & NoSQL Databases,*
https://doi.org/10.1007/978-3-658-24549-8

Cypher Cypher is a declarative query language for the graph-based Neo4j database.

Data architecture A data architecture determines which data an organization will gather on their own and which they will obtain from information brokers, which data is to be stored centralized or non-centralized, how data protection and data security are ensured, and who is responsible for maintenance and upkeep of the stored data.

Database schema A relational database schema formally specifies databases and tables, lists key and non-key attributes, and determines integrity constraints.

Database system A database system consists of a storage and a management component. The storage component is used to store data and relationships; the management component provides functions and language tools for data maintenance and management.

Data dictionary system Data dictionary systems are used for the description and documentation of data elements, database structures, transactions, etc. and their connections with each other.

Data independence Data independence in database management systems is established by separating the data from the application tools via system functionalities.

Data management Data management includes all operational, organizational, and technical aspects of data architecture, data governance, and data technology that support company-wide data storage, maintenance, and utilization as well as business analytics.

Data mining Data mining is the search for valuable information within data sets and aims to discover previously unknown data patterns.

Data model Data models provide a structured and formal description of the data and data relationships required for an information system.

Data protection Data protection is the prevention of unauthorized access to and use of data.

Data scientist Data scientists are business analytics specialists and experts on tools and methods for NoSQL databases, data mining, statistics, and the visualization of multi-dimensional connections within data.

Data security Data security includes all technical and organizational safeguards against the falsification, destruction, and loss of data.

Data stream A data stream is a continuous flow of digital data with a variable data rate (records per unit of time). Data in a data stream is in chronological order and may include audio and video data or series of measurements.

Data warehouse A data warehouse is a system of databases and loading applications which provides historized data from various distributed data sets for data analysis via integration.

Document store Document stores, also called document-based databases, are NoSQL databases that store one record for each key, similar to key-value stores. However, those records contain sets of structured data in the form of attributes and characteristics, hence the name document.

End user End users are employees in the various company departments who work with the database and have basic IT knowledge.

Entity Entities are equivalent to real-world or abstract objects. They are characterized by attributes and grouped into entity sets.

Entity-relationship model The entity-relationship model is a data model defining data classes (entity sets) and relationship sets. In graphic representations, entity sets are depicted as rectangles, relationship sets as rhombi, and attributes as ovals.

Fuzzy database Fuzzy databases support incomplete, unclear, or imprecise information by employing fuzzy logic.

Generalization Generalization is the abstraction process of combining entity sets into a superordinate entity set. The entity subsets in a generalization hierarchy are called specializations.

Graph database Graph databases manage graphs consisting of vertices representing objects or concepts and edges representing the relationships between them. Both vertices and edges can have attributes.

Graph-based model The graph-based model represents real-world and abstract information as vertices (objects) and edges (relationships between objects). Both vertices and edges can have properties, and edges can be either directed or undirected.

Hashing Hashing is a distributed storage organization in which the storage location of the data records is calculated directly from the keys using a transformation (hash function).

Index An index is a physical data structure that provides the internal addresses of the records for selected attributes.

In-memory database In in-memory databases, the records are stored in the computer's main memory.

Integrity constraint Integrity constraints are formal specifications for keys, attributes, and domains. They ensure the consistent and non-contradictory nature of the data.

Join A join is a database operation that combines two tables via a shared attribute and creates a result table.

Key A key is a minimal attribute combination that uniquely identifies records within a database.

Key-value store Key-value stores are NoSQL databases in which data is stored as key-value pairs.

MapReduce method The MapReduce method consists of two phases: During the map phase, subtasks are delegated to various nodes of the computer network in order to use parallelism for the calculation of preliminary results. Those results are then consolidated in the reduce phase.

Normal form Normal forms are rules to expose dependencies within tables in order to avoid redundant information and resulting anomalies.

NoSQL NoSQL is short for 'Not only SQL' and describes databases supporting Big Data that are not subject to a fixed database schema. In addition, the underlying database model is not relational.

NULL value A NULL value is a data value that is not yet known to the database.

Object-orientation In object-oriented methods, data is encapsulated by appropriate means and properties of data classes can be inherited.

Optimization The optimization of a database query comprises the rephrasing of the respective expression (e.g. algebraic optimization) and the utilization of storage and access structures to reduce the computational expense.

QBE QBE (Query by Example) is a database language in which users have their intended analyses created and executed based on examples.

Query language Query languages are used to analyze and utilize databases, potentially set-orientedly, via the definition of selection conditions.

Recovery Recovery is the restoration of a correct database state after an error.

Redundancy Multiple records with the same information in one database are considered redundancies.

Relational algebra Relational algebra provides the formal framework for the relational database languages and includes the set union, set difference, Cartesian product, project, and select operators.

Relational calculus Relational calculus is based on propositional logic, with quantifiers ("for all …" or "there exists …") being permitted in addition to logical connectives between predicates.

Relational model The relational model is a data model that represents both data and relationships between data as tables.

Selection Selection is a database operation that yields all tuples from a table that match the criteria specified by the user.

SQL SQL (Structured Query Language) is the most important relational query and manipulation language and has been standardized by ISO (International Organization for Standardization).

Table A table (also called relation) is a set of tuples (records) of certain attribute categories, with one attribute or attribute combination uniquely identifying the tuples within the table.

Transaction A transaction is a sequence of operations that is atomic, consistent, isolated, and durable. Transaction management allows conflict-free simultaneous work by multiple users.

Tree A tree is a data structure in which every node apart from the root node has exactly one previous node and where there is a single path from each leaf to the root.

Two-phase locking protocol The two-phase locking (2PL) protocol prohibits transactions from acquiring a new lock after a lock on another database object used by the transaction has already been released.

Vector clock Vector clocks are no time-keeping tools, but counting algorithms allowing for a partial chronological ordering of events in concurrent processes.

XML XML (eXtensible Markup Language) describes semi-structured data, content, and form in a hierarchical manner.

References

Bachmann, R., Kemper, G., Gerzer, T.: Big Data—Fluch oder Segen? Unternehmen im Spiegel des gesellschaftlichen Wandels. mitp, Heidelberg (2014)

Davenport, T.H.: Big Data at Work—Dispelling the Myths, Uncovering the Opportunities. Harvard Business School Press, Boston (2014)

Fasel, D., Meier, A. (Hrsg.): Big Data. Praxis der Wirtschaftsinform. **51**(4), 298 (2014)

Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A.H.: Big Data—The Next Frontier for Innovation, Competition, and Productivity. McKinsey Global Institute, Washington (2011)

Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O'Reilly and Associates, Cambridge (2013)

Schwarz, R., Mattern, F.: Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Distrib. Comput. **3**(7), 149–174 (1994)

Shamos, M.I.: Computational Geometry. PhD Thesis, Department of Computer Science, Yale University, New Haven, USA (1987)

Silverston, L.: The Data Model Resource Book, vol. 2. Wiley, New York (2001)

Van Bruggen, R.: Learning Neo4j. Packt, Birmingham (2014)

Van Stehen, M.: Graph Theory and Complex Networks—An Introduction. Maarten van Stehen, Amsterdam (2010)

A. Meier and M. Kaufmann, *SQL & NoSQL Databases,*
https://doi.org/10.1007/978-3-658-24549-8

Index

A. Meier and M. Kaufmann, *SQL & NoSQL Databases,*
https://doi.org/10.1007/978-3-658-24549-8