Information Systems

Foundations, Third Class

Contents

1 Introduction

Databases today are essential to every business. Whenever we visit a Web site that provides information such as Amazon or Google there is a database behind serving up the information we request. Companies or researchers maintain their data in databases as well. As databases have been developed over several decades they are quite powerful. A specialised software called a database management system (DBMS), enables to create and manage large amounts of data and allows it to persist over long periods of time. [3]

1.1 Why Databases?

At the heart of modern information systems are the collection, storage, aggregation, manipulation, dissemination and management of data. Depending on the type of information system, these data vary from some megabytes to several terabytes. It is estimated that Google responds to more than 5.74 billion searches per day across a collection of data that is about several terabytes. How is it possible that the results of these searches are retrieved almost instantly? The answer is that they use databases. [1]

1.1.1 Data vs. Information

Important to know when designing databases is the difference between data and information.

"**Data** are raw facts." It is assumed that data can be recorded and stored on computer media. Examples are a telephone number, a birth date or a student name. Data have little meaning unless they have been organised in some logical order.

"**Information** is the result of processing raw data to reveal its meaning." Information is produced by processing data. Such information can be used as the foundation for decision making. Timely and useful information requires accurate data.

1.1.2 Introducing the Database

A **database** is a shared, integrated computer structure that stores a collection of:

- End-user data, or raw facts of interest to the end user.
- **Metadata** or data about data ... through which end-user data is integrated and managed.

The metadata provide a description of data characteristics and relationships that link the data within the database. In common, the term **database** refers to a collection of data that is managed by a DBMS [2].

A database management system (DBMS) is a collection of programs that manages the database structure and controls access to the data stored in the database [1].

Role and Advantages of the DBMS The DBMS serves as the intermediary between the end user and the database (see also Figure 1.1).

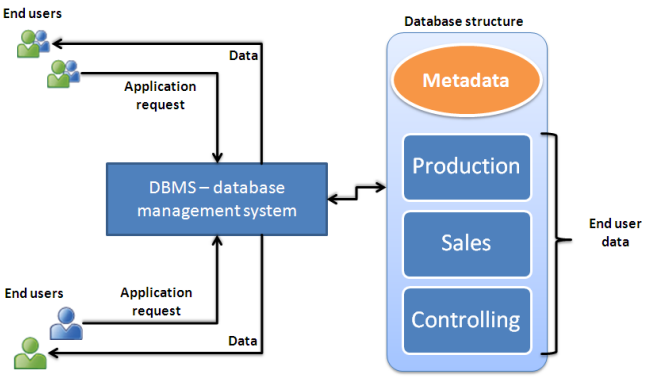


Figure 1.1: The DBMS as the intermediary between the end user and the database (according to [1])

The DBMS is expected to [2]:

- Allow users to create new databases and specify their schemas (logical structure of the data), using a specialised *data-definition language*.
- Allow users to *query* the data and modify the data, using an appropriate *query language* or *data-manipulation language*.
- Allow users to store very large amounts of data over a long period of time, supporting efficient access to the data for queries and database modifications.
- Enable *durability*, the recovery of the database in the face of failures, errors of many kinds or intentional misuse.
- Control access to data from many users at the same time, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

The DBMS provides advantages such as [1]:

- *Improved data sharing*. The DBMS enables end users to get better access to more and better-managed data.
- *Improved data security.* The DBMS provides a framework for better enforcement of data privacy and security.
- *Minimized data inconsistency.* **Data inconsistency** exists when different versions of the same data appear in different places. A properly designed database reduces the probability of data inconsistency.
- *Improved data access*. The DBMS allows to produce quick answers to ad hoc queries. A **query** is a specific request sent to the DBMS for data manipulation - like a question about the data, whereby **ad hoc** means on the spur of the moment. The answer given by the DBMS is called **query result set**.
- *Improved decision making*. Better-managed data and data access provide better quality information, on which further decisions can be based. However, the quality of information depends on the quality of the stored data. **Data quality** is an approach to promote the accuracy, validity and timeliness of the data.
- *Increased end-user productivity*. Available data and tools, which transform data into useful information, allow end users to react more quickly when dealing in the global economy.

Types of Databases A DBMS can support different types of databases. Databases can be classfied according to the number of users, the database location and the type of use [1].

A **single-user database** is restricted to one user at a time. If this single-user database runs on a personal computer it is called **desktop database**. A **multiuser database** servers several users at the same time. Depending on the number of users, this kind of database is either called **workgroup database** (fewer than 50 users) or **enterprise database** (more than 50 users, many departments).

A **centralised database** is located on a single site. In contrast, a **distributed database** is distributed across several different sites.

Depending on the information gathered from the databases, we can classify **general-purpose databases** or **discipline-specific databases**. A database that is designed to support a company's day-to-day operations is called **operational database** (or online transaction processing (OLTP), transactional, or production database). In contrast, an analytical database is dedicated to store data that is used to generate information required to make tatical or strategic decisions. It comprises two main components: a data warehouse and an online analytical processing front end. The **data warehouse** stores data in a format optimised for decision support. It hosts historical data as well as data from external sources. **Online analytical processing** (OLAP) is a set of tools that enables advanced data analysis from the underlying data within the data warehouse.

1.1.3 Historical Roots: Files and File Systems

The first commercial database management systems were available in the late 1960's. These systems based on file systems, which store large amounts of data over a long period of time in seperate files. Organisations needed these systems to handle core business tasks. Historically, such systems were often manual systems. This means that it was written on paper how to use the data within these file folders.

However, generating reports from manual file systems was slow and exhausting. Consequently, the organisations looked for data processing specialists who created computer based systems (application programs) that tracked data and produced required reports. Whenever managers wanted data from the file system they had to send a request to the data processing specialist.

Figure **??** shows an example for using single, isolated files for "products". Each application program uses an isolated file to store the data requested by the organisational section.

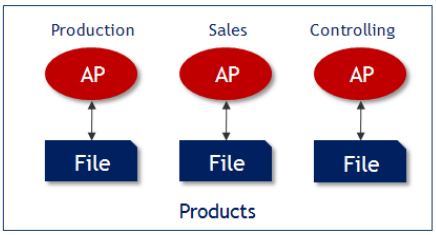


Figure 1.2: Example "Products" - Isolated Files

Figure 1.3 shows an example for using one single file for several application programs to manage the data of "products".

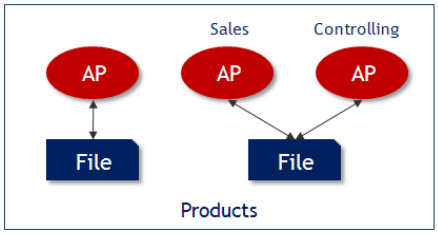


Figure 1.3: Example "Products" - Shared Files

The problems with file systems evolved from the increasing amount of computerised files. Having many data files that contained related, often redundant data with no means of controlling the data across all of the files, provoked many difficulties. In the following, main problems when using file systems for data processing are described[1]:

- *Lengthy development times.* Each data-retrieval task - no matter how simple or complex - requires extensive programming. The programmer has to specify what has to be done and how to do it.
- *Difficulty of getting quick answers.* Ad hoc queries are not possible because of the need to write firstly a program to get information out of the file system.
- *Complex system administration.* The increasing amount of files makes it difficult for system administrators to keep the files up-to-date and without fault.
- *Lack of security* and *limited data sharing*. Sharing data among multiple groups hosts a lot of security risks, for instance, effective password protection, the ability to lock out parts of files or parts of the system.
- *Extensive programming*. Changing a single field in a file system environment may provoke several changes at different locations in the application programs.

A file system exhibits structural dependence. That means that the access to a file is dependent on its structure. In contrast, **structural independence** exists when it is possible to make changes in the file structure without affecting the application program's ability to access the data. [1]

A file system exhibits data dependence. That means that changes in file characteristics require changes in all programs that access the file. In contrast, **data independence** assumes that physical representation and location of data and the use of data are separated.

- *Logical data independence*. Protection from changes in logical structure of data.
- *Physical data independence.* Protection from changes in physical structure of data.

Data redundancy exists when the same data are stored at different places. Uncontrolled data redundancy leads to [1]:

- *Data inconsistency*. Data inconsistency exists when different and conflicting versions of the same data appear in different places. Data that display data inconsistency are also referred to as data that lack data integrity. **Data integrity** is defined as the condition in which all data in the database are consistent with real-world events and condtions [1]. It means that
  - Data are *accurate* - there are no data inconsistences.
  - Data are *verifiable* - the data will always yield consistent results.
- *Data anomalies*. A data anomaly develops when all required changes in the redundant data are not made successfully. The data anomalies are defined as following:
  - *Update anomalies* occur when changes are made to existing redundant data.
  - *Insertion anomalies* occur when entering new data.
  - *Deletion anomalies* occur when deleting existing, dependent data.

1.1.4 Database Systems

The problems with file systems make using a database system very desirable. A **database system** refers to an organisation of components that define and regulate the collection, storage, management and use of data within a database environment [1].

One big benefit is that unlike the file system, the database consists of logically related data stored in a single logical data repository (as shown in Figure 1.4).

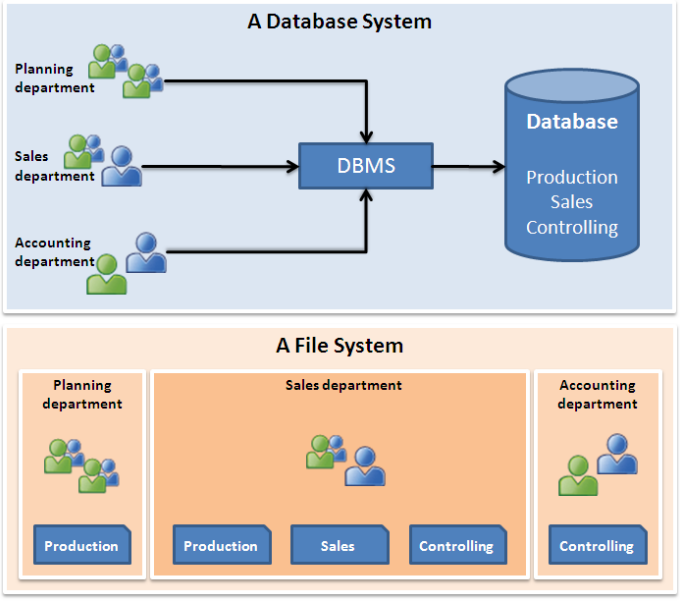


Figure 1.4: Database and file system in contrast (according to [1])

The database's DBMS provides numerous advantages (as already mentioned in 1.1.2). It stores not only the data structures but also the relationships between those structures. It provides access paths to these structures in a central location.

From a management point of view, the database system can be described by five components [1]:

- *Hardware*. Hardware compromises all of the system's physical devices, e.g. workstations, servers, storage devices, printers, network devices, ..
- *Software.* There are three types of software which are needed:
  - *Operating system software* such as Microsoft Windows, Linux, Mac OS or UNIX allows all other system to run on the computers.
  - *DBMS software* manages the database within the database system. DBMS software vendors are for instance Microsoft (Access and SQL Server), Oracle or IBM (DB2).
  - *Application programs* and *utility software* are used to access and manipulate data in the database. Application programs are mostly used to generate reports or information for facilitating decision making. Utilities are software tools that help to manage the components of the database system, e.g. to create database structures, control database access or monitor database operations.
- *People*. Dependent on the person's role there are following types of users in a database system:
  - *System administrators* manage the general operations of the database system.
  - *Database administrators* manage the DBMS and ensure proper database functionality.
  - *Database designers* are the database architects. They design the database structure.
  - *System analysts* and *programmers* design and implement the application programs.
  - *End users* use the application programs to do their daily operations.
- *Procedures*. Procedures are used to build instructions and rules that govern the use of the database system.
- *Data*. Data covers the collection of facts that is stored in the database.

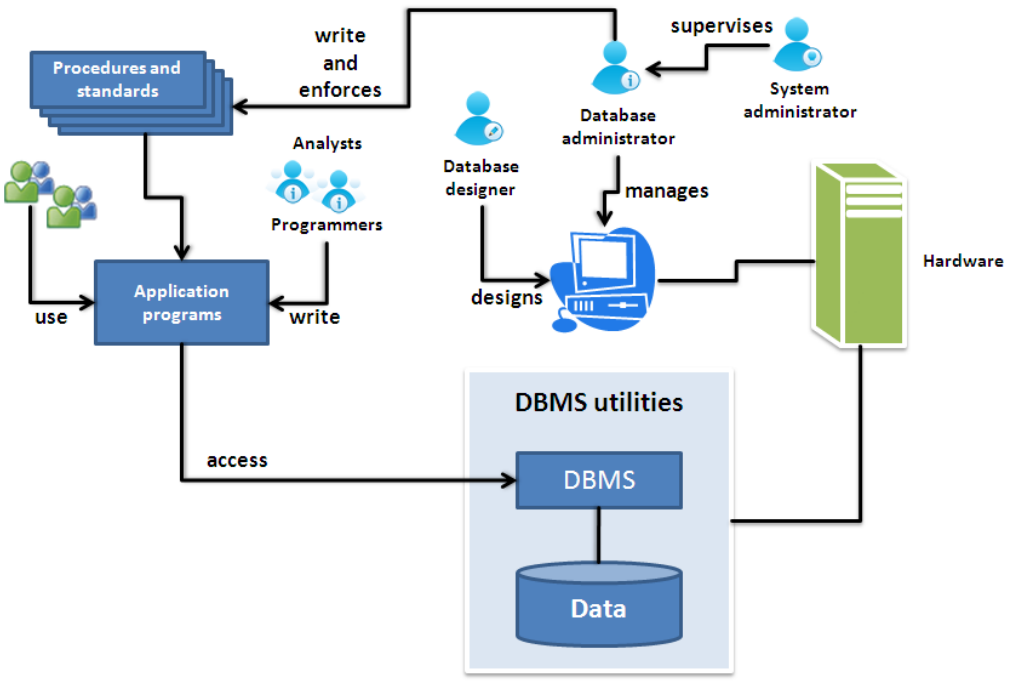


Figure 1.5: The database system environment (according to [1])

1.2 Data Models

"A data model is a relatively simple representation, usually graphical, of more complex real-world data structures." Besides, the terms data model and database model are often used interchangeably [1]. The representation generally consists of three parts [2]:

- *Structure of the data.* The data structures which are used to implement data in the computer - also called a physical data model.
- *Operations on the data.* In database models there is a limited set of operations that can be performed: queries (operations that retrieve information) and modifications (operations that change the database).
- *Constraints on the data.* Data models allow describing limitations on what the data can be. These constraints can be very simple such as "a month is an integer between 1 and 12" or very complex such as "if the student has not finalised course X he/she cannot attend course Z".

1.2.1 The Evolution of Data Models

In the following, we present data models that were developed to provide a way for specifying the structures of data. Today, most common data models are the relational model and the semistructured-data model, including the object-relational model and XML. However, new technologies subsumed under the term NoSQL emerge in the scope of science and Web affine companies.

Hierarchical and Network Models The hierarchical model was developed in the 1960s. It was dedicated to manage large amounts of data such as the data needed by the Apollo rocket that landed on the moon in 1969. The basic logical structure is represented by an upside-down tree. It contains levels and segments. A segment is similar to a record type within a file system. It depicts a one-to-many relationship between a parent and its children segments. Consequently, one parent can have several children, whereby each child belongs to one parent.

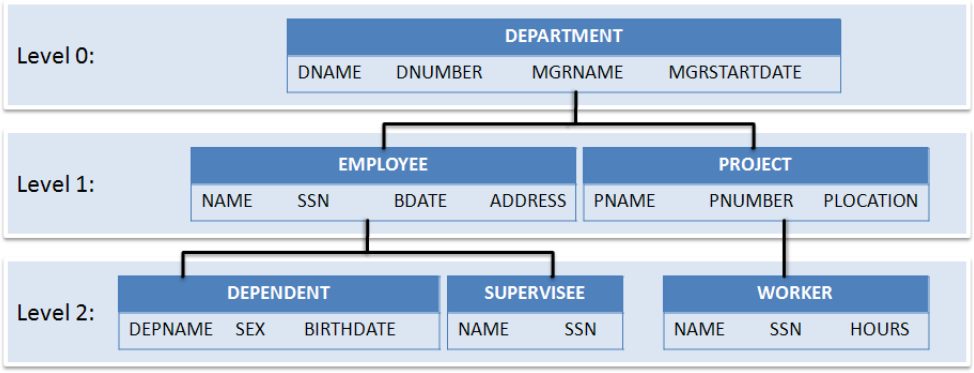


Figure 1.6: An example hierarchical data model (according to [4])

The network model allows representing complex data relationships more effectively. Similar data is stored in record sets, which are in relationship. A record set is composed by at least two record types: an owner and a member. The owner is an equivalent to the hierarchical model's parent. The member is an equivalent to the hierarchical model's child.

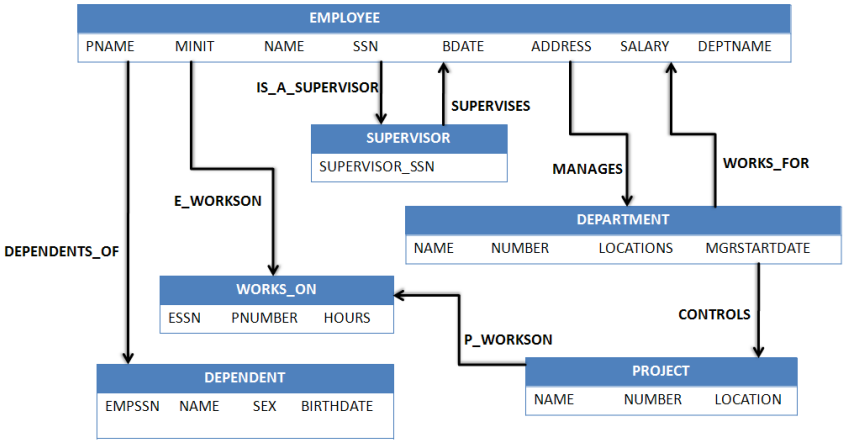


Figure 1.7: An example network data model (according to [4])

The Relational Model The relational model is based on tables.

title	year	length	genre
Star wars	1977	124	sciFi
Gone with the Wind	1939	231	drama
Wayne's World	1992	95	comedy

Figure 1.8: An example relation

The Object-Oriented (OO) Model The object-oriented model allows modelling more complex real-world scenarios.

Object-Relational and XML The object-relational model allows adding object-oriented features to the relational model. This object-orientation on relations has two effects: values can have structures - rather than single types and relations can have associated methods.

Semistructured-data models such as XML resemble trees or graphs rather than tables.

```
<movies>
  <movie title="Star wars">
    <year>1977</year>
    <length>124</length>
    <genre>sciFi</genre>
  </movie>
  <movie title="Gone with the Wind">
    <year>1939</year>
    <length>231</length>
    <genre>drama</genre>
  </movie>
  <movie title="Wayne's World">
    <year>1992</year>
    <length>95</length>
    <genre>comedy</genre>
  </movie>
</movies>
```

Figure 1.9: Movie data as XML

NoSQL (Not only SQL) The term NoSQL is "used to describe the increasing usage of non-relational databases among Web developers".

1.2.2 Degrees of Data Abstractions - The Three-tier Architecture

According to the three-tier Architecture by ANSI/SPARC developed in 1978, each view describes an aspect of the database relevant to a particular group of users.

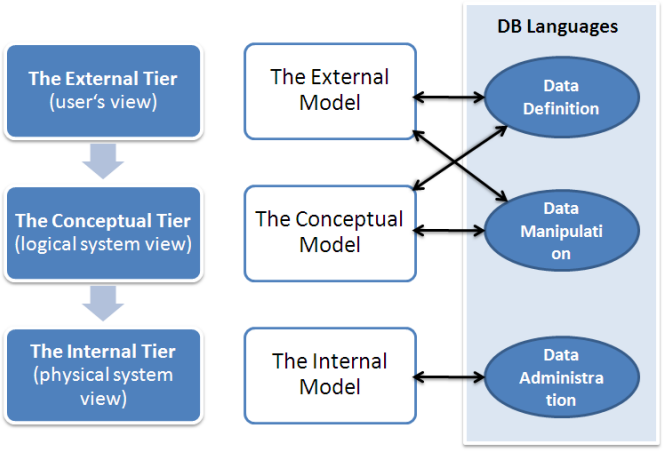


Figure 1.10: The three-tier architecture by ANSI/SPARC 1978

The External Model (user's view) The external model describes the part of the database that a particular user group, e.g. sales department, controlling department, is interested in. This model represents **parts of the conceptual model**. These parts are also called views.

The Conceptual Model (logical system view) The conceptual model describes the structure of the whole database for a community of users. This model describes **which data** is stored in the database. It corresponds to the information that is available for a particular domain, e.g. customers with customer name, address; products with name, weight, type or price.

The Internal Model (physical system view) The internal model determines the physical storage structure of the database (e.g. formats of physical records, access path). It defines **how the data is stored** physically on disk, in buffers, in blocks, with/without indexes.

With respect to these three models data independency as already mentioned in Section 1.1.3 can be provided:

- *Logical data independence* between the external and conceptual tier.
- *Physical data independence* between the internal and conceptual tier.

1.3 Overview of a Database Management System [2]

Figure 1.11 illustrates an outline of a complete DBMS. Single boxes represent system components, double boxes represent in-memory data structures, solid lines indicate control and data flow, and dashed lines indicate data flow only. We suggest that there are two sources of commands to the DBMS:

1. Conventional users and application programs ... they ask for data or modify data.
2. A database administrator (DBA) ... a person who is responsible for the structure or schema of the database.

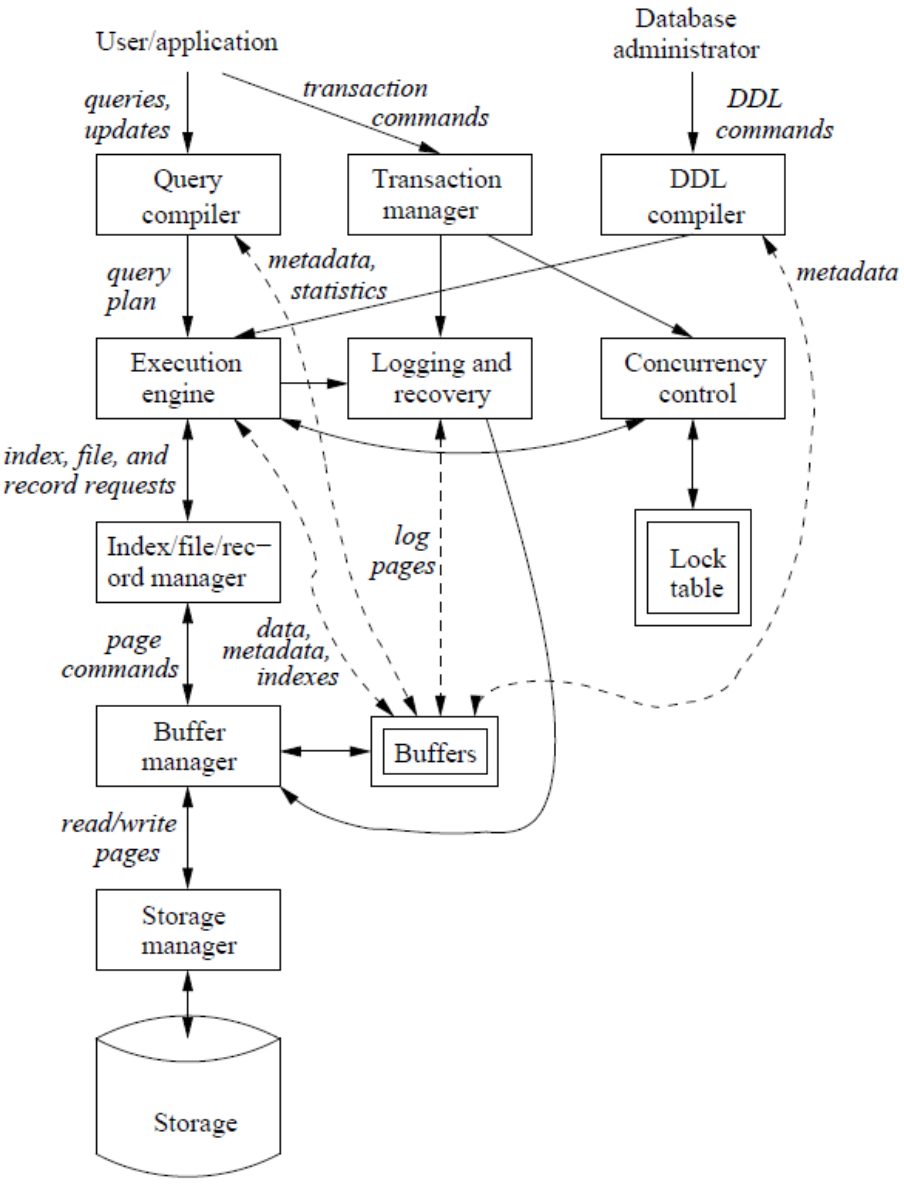


Figure 1.11: Database management system components

1.3.1 Data-Definition Language Commands

The schema-altering data definition language (DDL) allows commands to define and manipulate the schema (structure and constraint information) of the database. These commands are parsed by a DDL processor and passed to the execution engine. The execution engine handles over to the index/file/record manager to alter the metadata. For performing DDL commands one has to get special authority, for instance, a database administrator.

The DBMS stores definitions of the data elements and their relationships (metadata) in a **data dictionary**. The DBMS uses the data dictionary to look up the data structures and relationships. Any changes made in the database structure are automatically recorded in the data dictionary without influencing application programs. The DBMS provides data abstraction and removes structural and data dependency from the systems.

1.3.2 Query Processing

A user or an application program uses the data manipulation language (DML) to perform some actions. This command does not affect the schema of the database but only the data. DML statements are executed by two

subsystems:

- *Answering the query.* (see Section 1.3.5)
- *Transaction processing.* (see Section 1.3.4)

1.3.3 Storage and Buffer Management

The storage manager is responsible for deciding which data must be in main memory to perform any useful operation on data. It keeps track of the location of files on the disk and obtains the blocks on request from the buffer manager.

The buffer manager is responsible for partitioning the available main memory into buffers. A buffer is a page-sized region into which disk blocks can be transferred. Information that may be requested by components are:

- *Data* means the content of the database.
- *Metadata* means the database schema that describes the structure of the database.
- *Log Records* give information about recent changes to the database.
- *Statistics* give information gathered and stored by the DBMS.
- *Indexes* are data structures that support efficient access to the data.

commit - vollständingen
rollback - zurück

1.3.4 Transaction Processing

Database operations are executed with the help of transactions. A **transaction** is a unit of work that must be executed atomically and in apparent isolation from other transactions. Besides, a DBMS guarantees durability, what means that the work of a completed transaction will never be lost. The transaction manager accepts transaction commands from applications. Then, the transaction processor performs the following tasks:

- *Logging.* Every change in the database is logged on disk. The log manager writes the log in buffers. The recovery manager examines the log of changes and restores the database in case of troubles.
- *Concurrency control.* Transactions must appear to execute in isolation. However, many transactions are executed in parallel. Consequently, the concurrency control manager (or scheduler) ensures that multiple transactions can be executed in an order that seems as one-at-a-time. Typically, the scheduler locks certain parts of the database.
- *Deadlock resolution.* The transaction manager is responsible for solving deadlock situations ("rollback" or "abort"). A deadlock situation happens when one transaction waits for resources from another and vice versa.

The ACID Properties of Transactions Properly implemented transactions meet the "ACID" test, where

- "A" stands for "atomicity", the all-or-nothing execution of transactions.
- "C" stands for "consistency", all databases have consistency constraints.
- "I" stands for "isolation", each transaction seems to be executed as no other transaction is executed at the same time.
- "D" stands for "durability", the effects on the database of a transaction will never get lost.

1.3.5 The Query Processor

The query processor determines the DBMS's performance. It is represented by two components:

- The *query compiler*, which translates the query into an internal form so-called *query plan*. A query plan is a sequence of operations that has to be performed on the data. These operations base on the implementation of **relation algebra** operations. The query compiler hosts three important components:
  - A *query parser*, which transforms the textual form into a tree structure.
  - A *query preprocessor*, which performs semantic checks on the query and tree transformations.
  - A *query optimiser*, which transforms the initial query plan into the best available sequence of operations.
- The *execution engine* is responsible for executing each of the steps in the chosen query plan. It interacts with most of the components of the DBMS. It must get the data from the database into buffers. It must interact with the scheduler to avoid access to locked data. It has to ensure that all database changes are properly logged.

1.4 Analogy: Databases - Programming Languages

In the following, we provide a comparison between databases and programming languages.

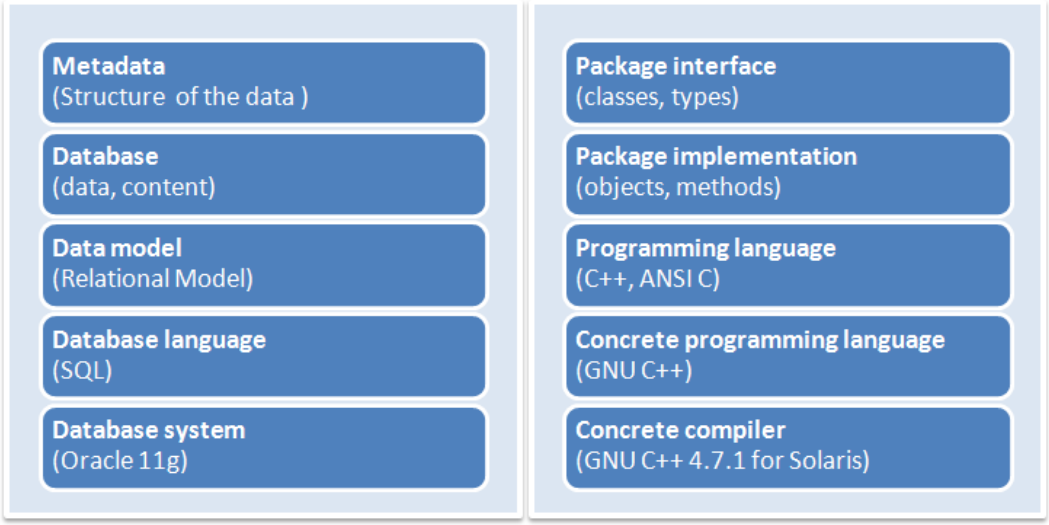


Figure 1.12: Analogy: Databases - Programming Languages

2 Database Design

When designing a database, the goals must be clearly defined in order to ensure that the development process runs smoothly from phase to phase. The primary goal of the end product has to be a database that meets the data storage needs of the customer with respect to performance and maintenance issues.

Database development is a systematic process that moves from concept to design to implementation. It also takes into account the needs of potential users and the operational and/or business processes in the organisation.

With database design you will get an answer to the following question: "How can we model real-world scenarios to provide a "good" database?"

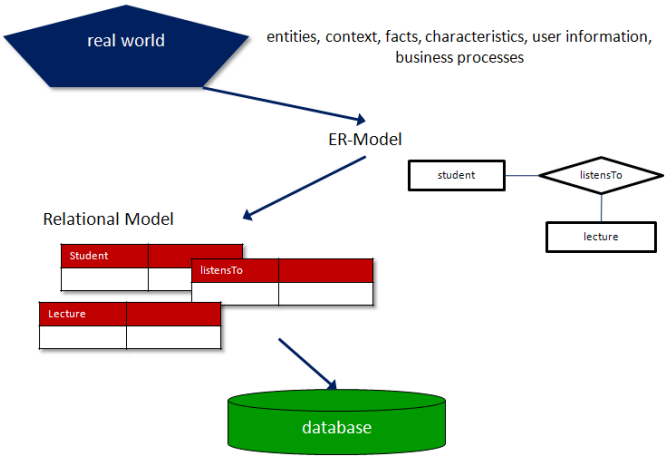


Figure 2.1: Database design - a simple overview

The database design process involves following phases:

- Requirements analysis
- Detailed System Design
  - Conceptual design
  - Logical design
  - Physical design
- Implementation
- Maintenance and evolution

These phases will be discussed in more detail in the following sections.

2.1 Requirements Analysis

It is the process of knowing and analysing the expectations of the users for the new database application in as much detail as possible. A team of analysts or requirement experts are responsible for carrying out the task of requirement analysis.

This phase determines what the users require. How these requirements will be implemented is of no importance in this phase.

Many database development efforts begin by defining the key business and/or operational processes within the organisation. Developers first create high-level models showing the major activity steps associated with marketing, sales, production, human resource management, public relations, research or development. Taken together, these process maps represent an enterprise-wide model of the organisation and its core processes.

Once a business process (or set of processes) has been selected, a further step is to define the information needs of users involved in or affected by the business process.

Course of actions

- Collection of relevant information such as statistical data, integrity constraints, user group analysis, available documents, questionnaires, interviews with users.
- Common understanding between the user and the developer (or designer).

Result

- Informal description about the real-world scenario, e.g. in form of text, tables, forms, business process maps.
- Separation of information dependent on data (data analysis) and functions (functional analysis), whereby data analysis is realised with the help of database design and functional analysis is performed by Software Engineering techniques.

2.2 Detailed System Design

After having collected user requirements, designers can go ahead and develop a detailed system design by means of the conceptual design, design of the logical data model and the physical design.

2.2.1 Develop conceptual design

A basic understanding of these needs is used to create a conceptual design for the database. At this stage, a conceptual data model is created that illustrates relationships between information sources, users and business process steps.

The conceptual design is a first formal description of the real-world scenario. It is dedicated to develop a platform-independent database schema, e.g. ER-Model, UML-Model.

Course of actions

- Modelling of different user views, e.g. for different departments.
- Analysing these views with respect to possible conflicts, e.g. naming, type or domain conflicts.

Result

- Conceptual design in form of an ER-Model or UML-Model.
- Communication means in form of an ER-Model or UML-Model for discussion between the user and the designer.

2.2.2 Develop logical data model

The conceptual data model is used to develop a logical data model based on one of the primary introduced types, for instance, relational, hierarchical, network, or object-oriented approaches.

The aim is to map data structures of the conceptual model correctly into corresponding structures in the logical data model, e.g. relations, XML-hierarchy.

Course of actions

- Transformation of the conceptual model, e.g. ER-Model into the logical model, e.g. Relational Model.
- In case that the relational model is used as logical model, the transformation has to be done with respect to rules concerning the normalisation of relations.

Result Logical schema, e.g. collection of relations.

2.2.3 Develop physical design

With the logical data model in hand, developers move to the physical design, which involves determining the specific storage and access methods and structures.

Course of actions

- Definition of the internal schema.
- Selection of useful storage structures.
- Definition of control access mechanisms aiming at minimising search pathes.

2.3 Implementation

Once this step is complete, developers can go ahead and create the database using whatever DBMS has been selected. Small amounts of data can be entered into the database for testing purposes. It is much easier to revise and change the database during this testing phase, before all of the data have been entered. The term prototyping refers to the iterative process used to try different report formats and input screens to determine their suitability and effectiveness.

2.4 Maintenance and Evolution

After a database has been created and all objects and data have been added and are in use, there will be times when maintenance must be performed. For example, it is important to back up the database regularly. You may also need to create some new indexes to improve performance.

Additionally, in case that further user requirements emerge, the database has to be adopted. When expanding or changing the database, it is important to consider each design process phase from scratch (this means that all phases of the design process have to be considered).

2.5 Summary

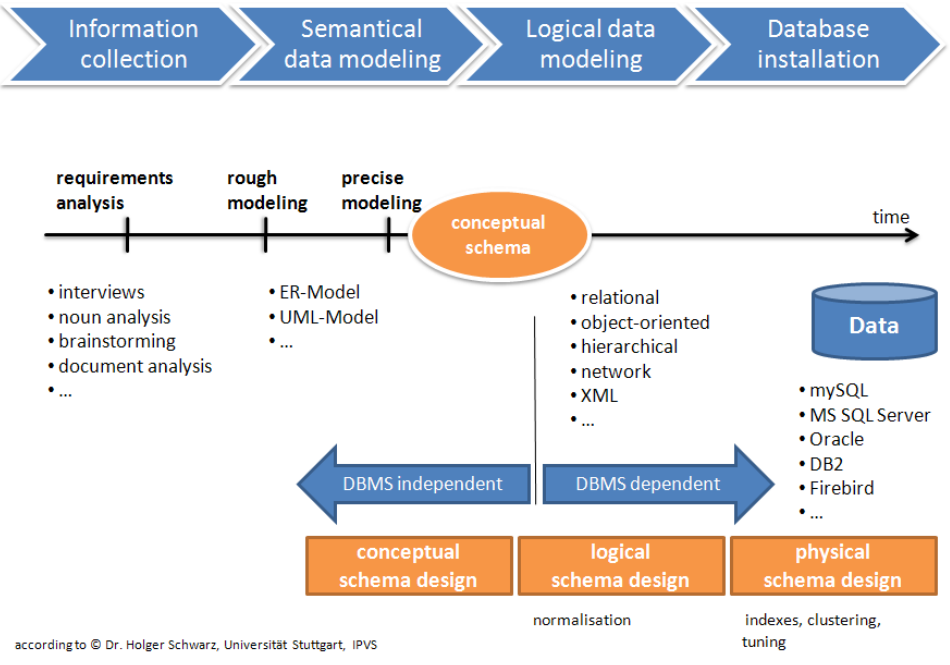


Figure 2.2: Database design process

3 Conceptual Design - Entity Relationship (ER) Modeling

P. Chen (1976): „The Entity-Relationship-Model – Toward a Unified View of Data“:

„This model incorporates some of the important semantic information about the real world. ... The entity-relationship model adopts the more natural view that the real world consists of entities and relationships.“

3.1 The Entity Relationship Model (ERM)

The ER model is used in the conceptual design phase. It describes data in terms of the following: entities, relationship between entities and attributes of entities:

- What are the entities and relationships in the enterprise?
- What information about these entities and relationships should be stored in the database?
- What are the integrity constraints or business rules that hold?
- A database schema in the ER model can be represented pictorially (ER diagrams).
- Can map an ER diagram into a relational schema.

In the following, we describe basic constructs, which are used for modeling an entity-relationship diagram.

3.1.1 Entities

An **entity** is represented as rectangle. An entity describes a real-world object distinguishable from other objects. An entity is described using a set of attributes. An entity can be a person, a place, an object, an event, or a concept about which an organisation wishes to maintain data.

An **entity type** defines a collection of entities that have same attributes. An **entity instance** is a single item in this collection. An **entity set** is a set of entity instances. Each entity set has a **key**. Each attribute has a **domain**.

3.1.2 Attributes

An **attribute** characterises existing entities or relationships in more detail. It is represented as ellipse or circle, which is connected to the associated entity type or relationship with lines.

There are actually several types of attributes. These include: simple, composite, single-valued, multi-valued, stored, and derived attributes.

- **Simple and composite attributes.** A simple or an atomic attribute, such as streetnumber or state, cannot be further divided into smaller components. A composite attribute can be divided into smaller subparts in which each subpart represents an independent attribute, e.g. name, address.
- **Single-valued and multi-valued attributes.** Single-valued attributes have a single value for an entity instance. A multi-valued attribute may have more than one value for an entity instance, e.g. language, which stores the names of the languages that a student speaks.
- **Stored and derived attributes.** The value of a derived attribute can be determined by analysing other attributes, e.g. age, which is calculated by date of birth and current date. A stored attribute describes a value that cannot be derived from the values of other attributes.
- **Key attribute.** A key attribute (or identifier) is a single attribute or a combination of attributes that uniquely identify an individual instance of an entity type.

3.1.3 Relationships

Logically linked entities are connected with relationships. A **relationship** as an association among several entities. A **relationship set** is a grouping of all matching **relationship instances**, and the term **relationship type** refers to the relationship between entity types.

A relationship type is represented with a diamond-shaped box (rhomb) connected by straight lines to the rectangles that represent participating entity types. A relationship type is a given name that is displayed in this diamond-shaped box and typically takes the form of a present tense verb or verb phrase that describes the relationship. Relationships are relevant for both sides. An example: Students attend courses; courses are attended by students.

The number (unary, binary, or ternary) of entity sets that participate in a relationship is called the **degree of relationship**.

Let E_1, E_2, ..., E_n denote n entity sets and let R be the relationship. Then, the degree of the relationship can be expressed as follows:

- **Unary relationship** (degree 1). An unary relationship R is an association between two instances of the same entity type (e.g., $R \in E_1 \times E_1$).
- **Binary relationship** (degree 2). A binary relationship R is an association between two instances of two different entity types (e.g., $R \in E_1 \times E_2$).
- **Ternary relationship** (degree 3). A ternary relationship R is an association between three instances of three different entity types (e.g., $R \in E_1 \times E_2 \times E_3$).
- An **n-ary relationship** describes relationships among several entity types. In most cases these relationships are characterised by attributes as well.

3.1.4 Connectivity and Cardinality

A cardinality is represented as letter or number. There are three kinds of cardinalities (it is supposed that there exist two entities A and B):

- **1:1-relationship** (spoken: one-to-one-relationship). One instance of entity B can be associated with a given instance of entity A and vice versa.
- **1:N-relationship** (spoken: one-to-many-relationship) or **N:1-relationship**. Many instances of entity B can be associated with a given instance of entity A. However, only one instance of entity A can be associated with a given instance of entity B. N:1-relationship is vice versa.
- **N:M-relationship** (spoken: many-to-many-relationship). Many instances of entity A can be associated with a given instance of entity B. Many instances of entity B can be associated with a given instance of entity A as well.

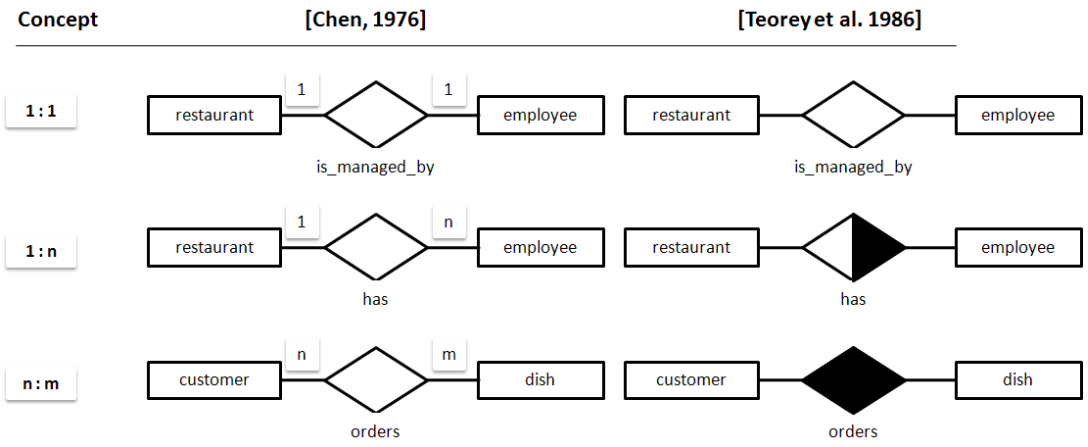


Figure 3.1: Basic relationship constructs

The cardinality of relationship represents the minimum/maximum number of instances of an entity that must/can be associated with any instance of another entity.

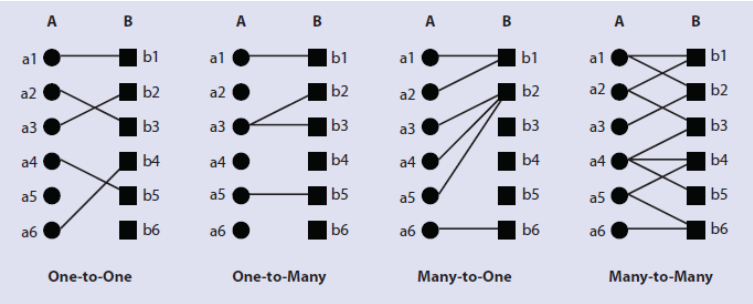


Figure 3.2: Cardinality of relationship

The cardinality is usually given by 1, N or M (uppercase or lowercase). The following diagram has to be interpreted as follows:

- One entity from E_1 can be associated with at most y entities from E_2.
- One entity from E_2 can be associated with at most x entities from E_1.

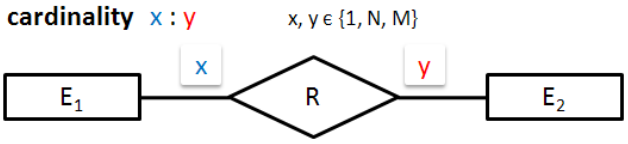


Figure 3.3: Cardinality of relationship

3.1.5 The (min, max) - Notation

The ER model introduces the (min, max) notation to specify an interval of possible participations in a relationship.

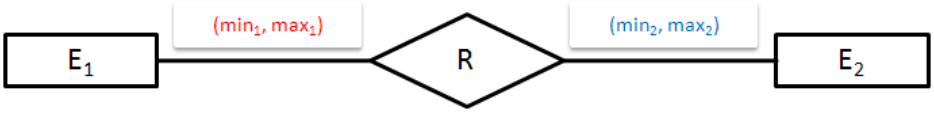


Figure 3.4: (min, max) Notation

- An entity of type E_1 may be related to at least min_1 and at most max_1 entities of type E_2 .
- Likewise, min_2 is the minimum number and max_2 is the maximum number of E_1 entities to which an E_2 entity is related.

Some extensions:

- "*" may be used as maximum if there is **no limit**.
- (0, *) means **no restriction** at all (general relationship).
- Normally, the minimum cardinality will be 0 or 1 and the maximum cardinality will be 1 or *. So, only the (0, 1), (1, 1), (0,*), (1,*) cardinalities are common in practice.

3.1.6 Weak Entities

Entity types can be classified into two categories: **strong entity types** and **weak entity types**. A strong entity type exists independent of other entity types, while a weak entity type depends on another entity type.

The entity type on which a weak entity type depends is called the **identifying owner** (or simply master), and the relationship between a weak entity type and its master is called an **identifying relationship**.

In such a case,

- there is a relationship with cardinality (1, 1) on the detail entity side and in addition
- the key of the master is inherited and becomes part of the key of the detail entity.

In ER diagrams, weak entities and their identifying relationships are indicated by double lines.

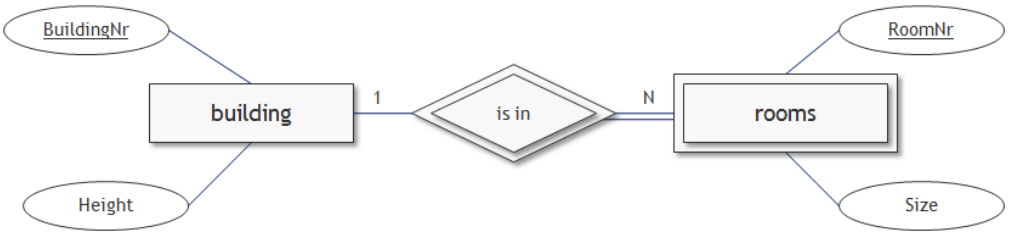


Figure 3.5: An Example: Weak Entity

3.1.7 ISA ('is a') Hierarchies

ISA hierarchies or relationships express either generalisations or specialisations. A generalisation describes less information whereby a specialisation goes more into the details.

An ISA relationship may have following characteristics:

- The relationship is either disjoint or not disjoint.
  - Disjoint means that none element of one subset can be find in another subset.
  - Not disjoint means that the subsets can contain common elements.
- The relationship is either total or partial.
  - Total means that there is no other subset according to this specialisation.
  - Partial means that there are normally other subsets according to this specialisation but listed in this context.

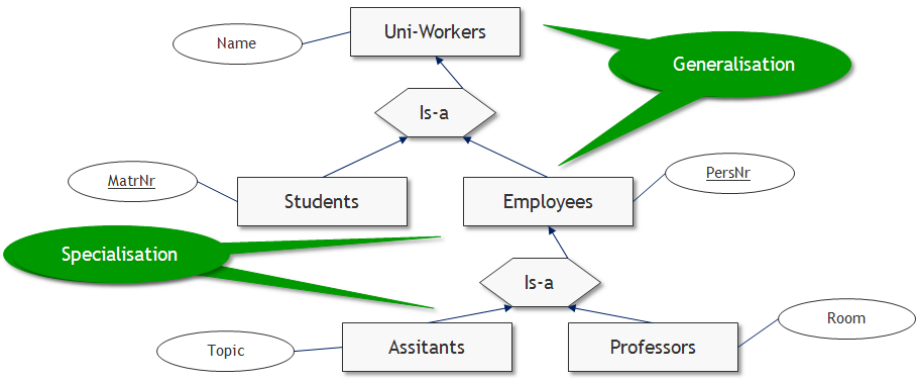


Figure 3.6: An Example: Specialisation and Generalisation

3.1.8 Aggregation

The aggregation is expressed by the part-of-relationship.

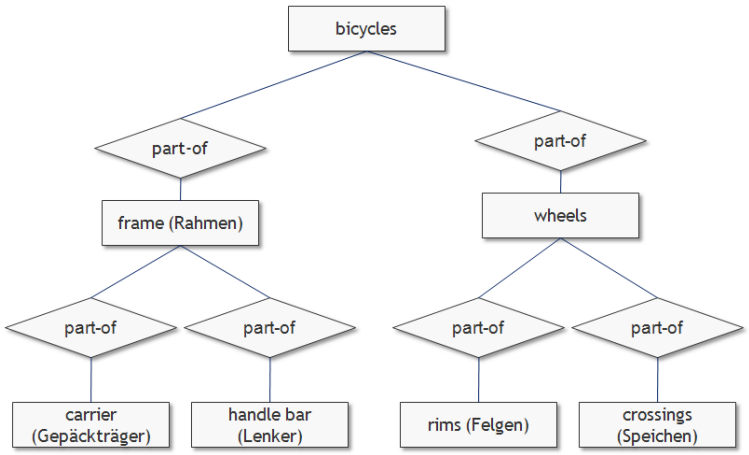


Figure 3.7: An Example: Aggregation

3.2 Developing an ER Model

When developing an ER Model there are some design choices that have to be considered. They are discussed in the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- Identifying relationships: binary or ternary? Aggregation?

3.2.1 Entity vs. Attribute

The decision for an entity or for an attribute is dependent on the use we want to make of a concept and the semantics of the data.

Example. Should *address* be an attribute of Employees or an entity (connected to Employees by a relationship)?

If we have **several** addresses per employee, *address* must be an entity (since there is no possibility that attributes are set-valued).

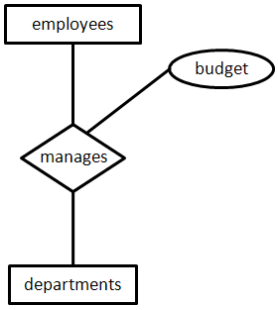
If the **structure** (street, city, postalcode, ...) is **important**, e.g. we search for employees living in a given city, *address* must be modeled as an entity.

3.2.2 Entity vs. Relationship

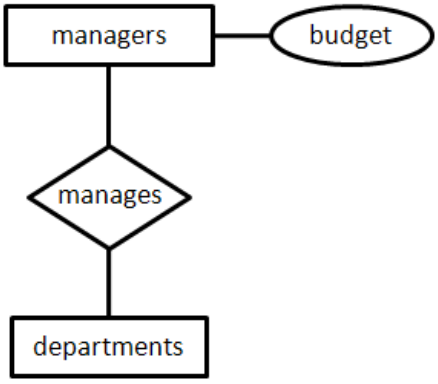
The decision for an entity or for a relationship is dependent on the use we want to make of a concept and the semantics of the data as well. A relationship should be used to describe an action where several entities are engaged.

Example. Should *manager/manages* be an entity or a relationship?

If a manager gets a separate budget for each department, *manages* can be modeled as relationship.



If a manager gets just a single budget to manage **all** departments, *manager* shall be modeled as entity.

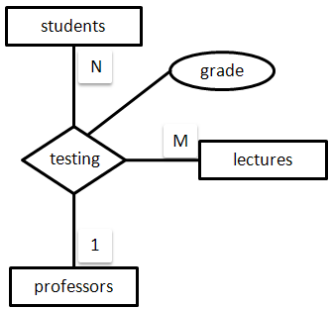
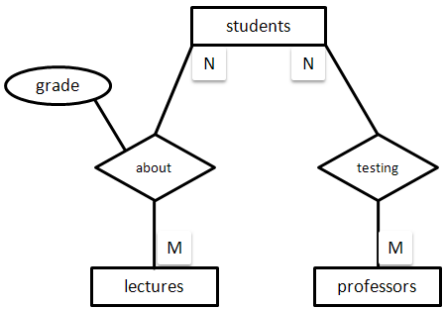


3.2.3 Binary vs. Ternary Relationships

The decision for a binary or a ternary relationship is dependent on the use we want to make of a concept and the semantics of the data as well.

Example. Should *testing* be modeled as binary or as ternary relationship?

If we want to model that a student is tested for a lecture by a professor, *testing* should be modeled as ternary relationship.

	
students × lectures → professors	No limitations - loose of information (If we introduce the attribute *timeofTesting* as attribute in both relationships such as *about* and *testing* we avoid loosing information. However, we have to check each time an exam is stored that these attributes are stored in both relationships as well.)

Modeling a ternary relationship with the help of several binary relationships may (dependent on the user requirements) have following drawbacks:

- You loose some information.
- You may generate inconsistent data. You have to consider consistency checks in the database.
- You model the real world inadequately.

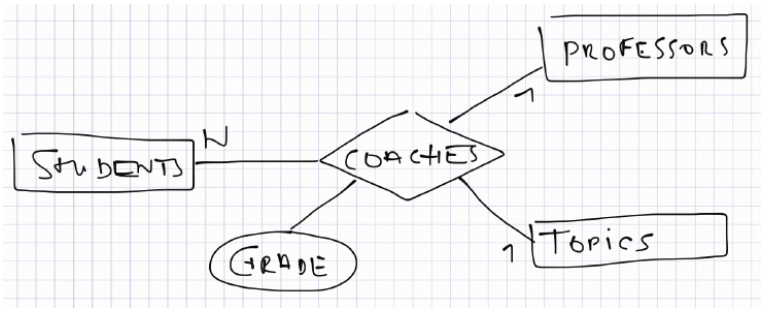


Figure 3.8: An Example: A tenary relationship

Modeling a tenary relationship - a further example. Following constraints (dt. Integritätsbedingungen) will be valid:

- Students are allowed to work on only one topic by the same professor.
- Students are allowed to work on one topic only once (even not by another professor).

Nevertheless, it is still possible that:

- Professors can assign a topic several times.
- A topic can be assigned by several professors - but to different students.

In summary:

- coaches: professors × students → topics
- coaches: topics × students → professors

Modeling a tenary relationship with the help of an entitytype instead of a relationship. Have a look at the following example:

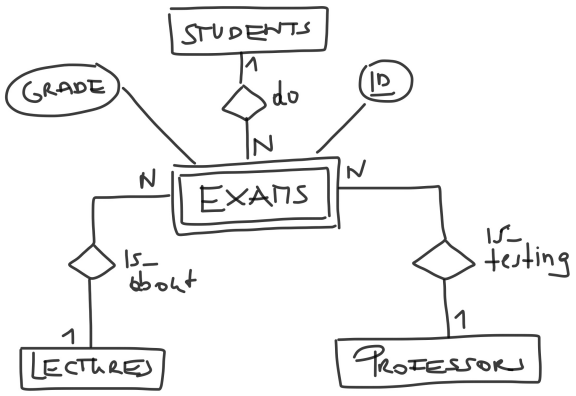


Figure 3.9: An Example: Entitytype instead of a relationship

It is possible that

- there are several professors for a single exam.
- several lectures will be tested in a single exam.

However, it is possible that an exam exists but there is no professor or the professor does not exist any more. To overcome this loss of semantic the (min-max)-notation may be used.

3.3 (min-max)-Notation

3.3.1 Introduction

In the following, (min, max)-relationships and functionalities (F_1, F_2) are illustrated:

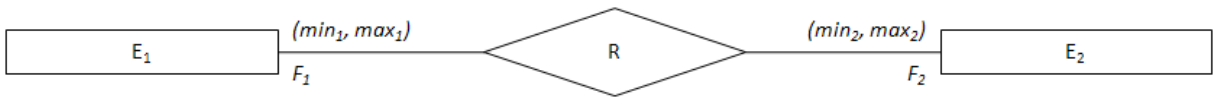


Figure 3.10: (min, max)-relationships and functionalities (F_1, F_2)

(Anmerkung: (min, max) erlaubt Unterscheidung, ob Teilnahme eines Entities an einer Beziehung optional (Mindestkardinalität 0) oder obligatorisch (Mindestkardinalität >= 1) ist. E_1 nimmt an (min_1, max_1) Beziehungen von Typ R teil. E_2 nimmt an (min_2, max_2) Beziehungen von Typ R teil.)

The connection between functionalities and (min, max)-notation is as follows:

$F_1 : F_2$	(min_1, max_1)	(min_2, max_2)
$1 : 1$	$(0, 1)$	$(0, 1)$
$1 : N$	$(0, *)$	$(0, 1)$
$N : 1$	$(0, 1)$	$(0, *)$
$N : M$	$(0, *)$	$(0, *)$

Example. Rivers can flow into the sea. Develop an ER-Diagram and show the functionalites in (F_1, F_2) and (min, max)-notation.

Solution. One river flows at most in one sea. In one sea flows at least one river, but normally more rivers.

(Ein Fluss mündet maximal in ein Meer. In ein Meer mündet mindestens ein Fluss, in der Regel aber mehrere Flüsse.)

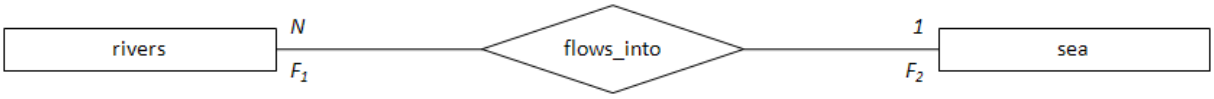


Figure 3.11: functionalities (F_1, F_2)

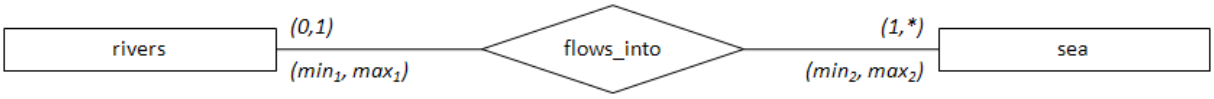


Figure 3.12: (min, max)-notation

Example. Interpret the functionality of the following ER-Diagram:

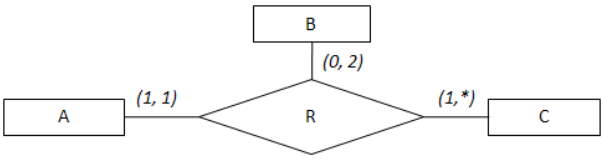


Figure 3.13: An Example: (min, max)-notation

Solution. In the relationship R

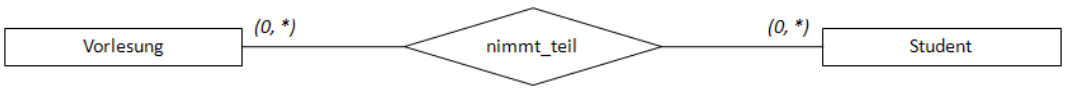
- each entity from A has to be present in exactly one tuple;
- an entity from B is allowed to be present in at most two tuples;
- each entity from C has to be present at least in one tuple.

(In R muss jedes Entity aus A in genau einem Tupel vertreten sein; darf ein Entity aus B höchstens zweimal in einem Tupel vertreten sein; muss jedes Entity aus C mindestens in einem Tupel vorkommen.)

3.3.2 Kombinationsmöglichkeiten und deren Interpretation

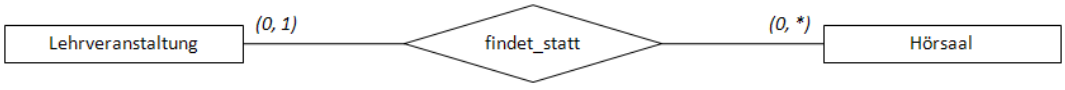
Kombinationen ohne Existenzabhängigkeiten. Kombinationen von Kardinalitäten implizieren keine Existenzabhängigkeit, wenn die Minimalkardinalität für alle Entitytypen, die an einer Beziehung teilnehmen, Null ist.

1. Kombination: (0, $*$) und (0, $*$)

   An einer Vorlesung nehmen *null bis beliebig viele* Studenten teil, Studenten können an *null bis beliebig vielen* Vorlesungen teilnehmen.

2. Kombination: (0, 1) und (0, $*$)

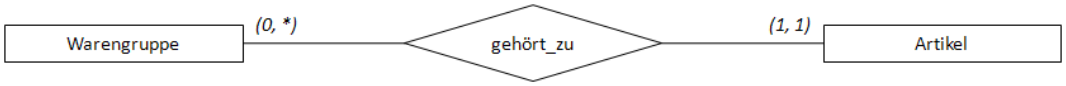
   Einer Lehrveranstaltung *kann maximal ein* Hörsaal zugeordnet werden (muss aber nicht: beispielsweise bei reinen Online-Veranstaltugen). In einem Hörsaal *können null bis beliebig viele* Vorlesungen stattfinden.

3. Kombination: (0, 1) und (0, 1)

Mitarbeitern kann *jeweils (maximal) ein PKW* als Dienstfahrzeug zugeordnet werden. Ein PKW *kann* der Dienstwagen von *(maximal) einem* Mitarbeiter sein.

Kombinationen mit einseitiger Existenzabhängigkeit. Kombinationen von Kardinalitäten implizieren eine einseitige Existenzabhängigkeit, wenn die Minimalkardinalität für genau einen an der Beziehung teilnehmenden Entityp Eins ist. Entities diesen Typs sind abhängig von den Entities, mit denen sie in Beziehung stehen.

1. Kombination: (0, $*$) und (1, 1)



Ein Artikel ist genau einer Warengruppe zugeordnet (d.h. es gibt auch keinen Artikel ohne Warengruppe). Warengruppen können null bis beliebig viele Artikel umfassen. Ein Artikel ist damit existenzabhängig von seiner Warengruppe, die Warengruppe ist jedoch unabhängig von ihren Artikeln.

2. Kombination: (0, $*$) und (1, $*$)

Ein Professor kann für null bis beliebig viele Vorlesungen als Dozent auftreten. Eine Vorlesung wird von mindestens einem (bis beliebig vielen) Professoren betreut. Vorlesungen sind damit existenzabhängig von Professoren.

3. Kombination: (0, 1) und (1, 1)

Eine Ärztin kann eine Station leiten (muss sie aber nicht), eine Station ist immer genau einer Ärztin zugeordnet. Damit ist die Station existenzabhängig von der Ärztin.

4. Kombination: (0, 1) und (1, $*$)

Spezialfall mit geringer praktischer Relevanz.

Kombinationen mit wechselseitiger Abhängigkeit. Wechselseitige Abhängigkeiten entstehen, wenn die Minimalkardinalität einer Beziehung auf beiden Seiten größer als Null ist.

1. Kombination: (1, $*$) und (1, 1)

Zu einer Rechnung existiert immer mindestens eine Rechnungsposition, eine Rechnungsposition ist genau einer Rechnung zugeordnet. Rechnung und Rechnungsposition sind wechselseitig voneinander abhängig.

2. Kombination: (1, $*$) und (1, $*$)

Spezialfall mit geringer praktischer Relevanz.

3. Kombination: (1, 1) und (1, 1)

Diese Kombination zeigt an, dass eine Beziegung immer zwischen exakt zwei Entities der verbundenen Entitytypen besteht. Dies impliziert zumeist, dass die beiden beteiligten Entitytypen zu einem Entitytyp zusammengefasst werden sollten.

4 Logical Design - The Relational Database Model

A relational database model is a DBS that is associated to the relational model.

The relational model was developed by Codd in 1970. It is based on predicate logic and set theory. **Predicate logic**, used in mathematics, provides a framework in which an assertion (statement of fact, i.G. Behauptung) can be verified as either true or false. **Set theory** is also used in mathematics and deals with sets or groups of objects. It is used as basis for data manipulation in the relational model.

Based on these concepts, the **relational model** has three well-defined components:

- how data are represented, namley through tables or relations (aspect of data structure). The user's view of data are tables.
- which rules these data have to fulfill (aspect of data integrity).
- how data can be manipulated (aspect of data manipulation). The user is provided by operators that allow him to create new from existing tables. There are at least operators such as *select* (extracts specific rows from a table and generates a new table), *project* (extracts specific columns from a table and generates a new table) and *join* (connects two tables based on same values in common columns).

The relational model represents data in a database as a set of relational tables (relations). A relation can be interpreted as table, where rows describe objects or relationships between objects. The name of the table is called relational table, the columns are called attributes and the rows are called tuples.

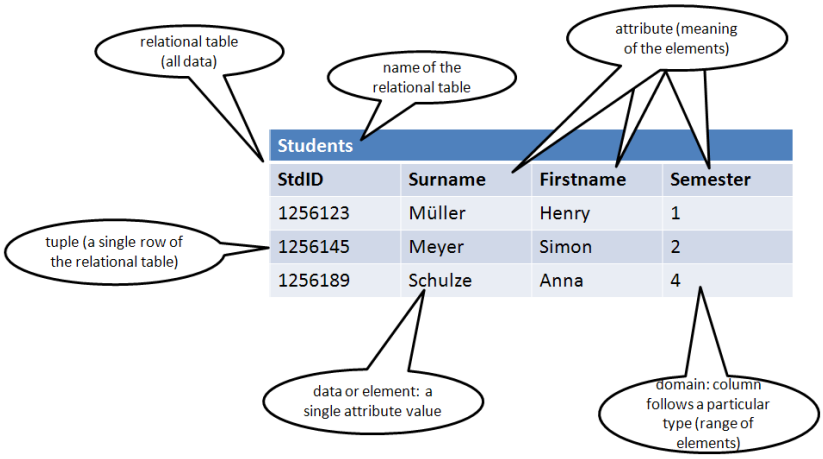


Figure 4.1: Naming convention of the relational model

A **relational table** has following characteristics:

- A table is perceived as a two-dimensional structure composed of rows and columns.
- The relational table of degree n has n columns.
- Each table row (tuple) represents a single entity occurence within the entity set.
- Each table column represents an attribute and each column has a distinct name.
- Each intersection of a row and column represents a single data value.
- All values in a column must conform to the same data format, so-called type integrity.
- Each column has a specific range of values known as the attribute domain.
- The order of the rows and the columns is immaterial to the DBMS.
- Each table must have an attribute or combinations of attributes that uniquely identifies each row, the so-called key integrity.

4.1 Formalisation

A *relation schema* R is a finite set of attribute names $A_1, A_2, ...A_n$. For each attribute name A_i there exists a set D_i, $1 \leq i \leq n$, which is called domain of A_i. It is also called as $\text{dom}(A_i)$.

A *relation* $r(R)$ that depends on a relational schema R is a finite set of references $\{t_1, t_2, ...t_m\}$ from R to D. These references are called *tuple*.

Example. Relational schema for RESTAURANT and DISH.

Restaurant = {rid, name, address, stars, type} Dish = {name, price, rid}

Example. Domains for the relational schema RESTAURANT.

dom(rid) = set of all integer. dom(name) = set of all names. dom(stars) = $\{k \mid 0 \leq k \leq 4\}$ (domain of stars equals the set of all k's such that k is greater or equal than zero and less than or equal than four)

Example. The relation RESTAURANT has more tuples. One of them is t with following values:

t(rid) = 3 t(name) = "Green Cottage" t(stars,type) = (2, "chinesisch") t = (3, "Green Cottage", "Kettenbrückengasse 3, 1050 Wien", 2, "chinesisch")

A *key* of a relational table $r(R)$ is a subset K of R. Then it is always true for two different tuples t_1 and t_2 of $r(R)$ that $t_1(K) \neq t_2(K)$ and that no real subset K' of K has this characteristic.

Commonly, one single key will be defined as primary key. The primary key will be characterised by underlining it in the relational schema.

Example. Characterising the key in the relational schema RESTAURANT and DISH.

Restaurant = {<u>rid</u>, name, address, stars, type} Speise = {<u>name</u>, price, <u>rid</u>}

4.2 Translating ER Models to Relational Models

Aiming at developing a relational DBS, an ER Model has to be translated into a relational model. This translation is performed through following steps:

- Translation of entities
- Translation of relationships
- Simplification of the schema

4.2.1 Translating Entities

Entity types are directly translated into relational tables. Each attribute of the entity will be transformed to an attribute of the relational table (key of the entity will be the key of the relational table).

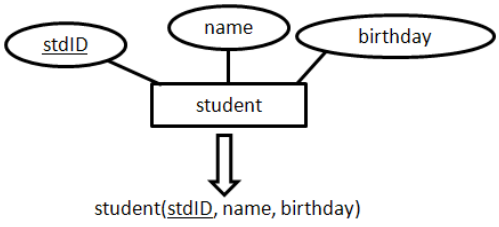


Figure 4.2: Example: Translating the entity *student*

4.2.2 Translating Relationships

In general, n-ary relationships are represented as follows in the ER Model.

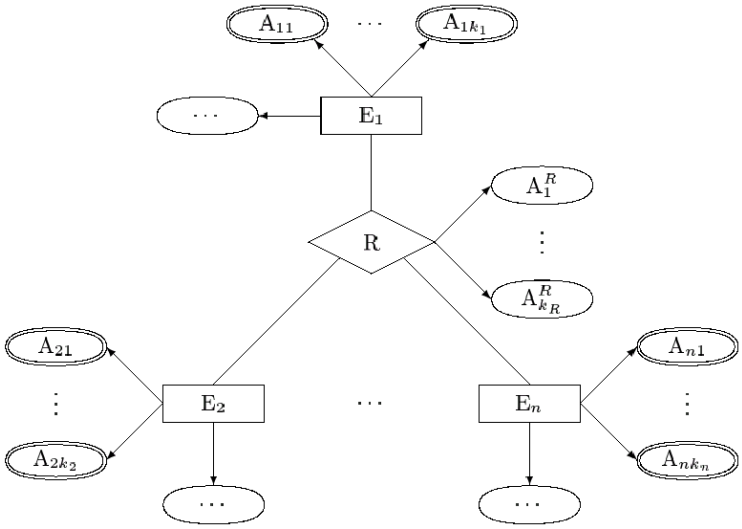


Figure 4.3: ER Model: n-ary relationships in general

This relationship is represented in the relational model as follows:

$$R = \{[\underbrace{A_{11}, ..., A_{1k_1}}_{\text{key of } E_1}, \underbrace{A_{21}, ..., A_{2k_2}}_{\text{key of } E_2}, ..., \underbrace{A_{n1}, ..., A_{nk_n}}_{\text{key of } E_n}, \underbrace{A_{R1}, ..., A_{Rk_R}}_{\text{attributes of } R}]\}$$

Subsequently, the relational table R hosts all key attributes of the entity types, which are part of the relationship, and the attributes of the relationship type as well.

The key attributes of the entity type are called **Foreign Keys** because they refer to tuples from other (foreign) relations. When integrating attributes from other entity types, it may be necessary to rename some of these attributes. The aim is that each attribute has a unique name.

One-to-one relationship (1:1)

Situation: Each entity of one type is related to exactly one entity of the other type and vice versa.

Translation: We can handle one-to-one relationships similar to one-to-many relationships. Important to consider is the "direction" of modeling. We prefer this translation that causes less NULL-values.

Entity types having the same primary keys may (but must not be) summarised to a single relational table.

Beispiel. Professors have an own room.

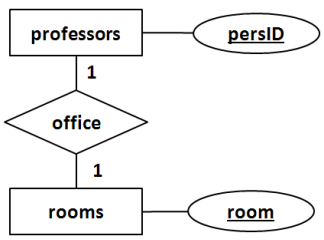


Abbildung 4.4: Example: 1:1 relationship

professors : {[persID: string; name: string; ...; **room**: string]}
rooms : {[room: string; size: double; ...]}

One-to-many relationship (1:N)

Situation: Each entity of one type can be related to several entites of the other type. However, each entity of the other type can only be related to at most one entity of the other type. (N:1 relationship is vice versa)

Translation: The primary key of the "1-side" will get foreign key of the "N-side".

Beispiel. Professors teach courses.

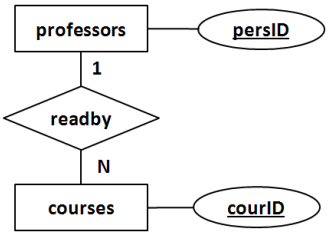


Abbildung 4.5: Example: 1:N relationship

courses : { [courID: string; title: string; semester: string; ...; **professorReading**: string] }
professors : {[persID: string; name: string; ...; **room**: string]}
The attribute *persID* has been renamed to the attribute *professorReading* in the relational table *courses* because of better readability.

Many-to-many relationship (N:M)

Situation: Each entity of one type can be related to several entities of the other type and vice versa.

Translation: These relationships are translated as n-ary relationships. Having a many-to-many relationship of two entity types will lead to three relational tables: two for the two entity types and a new one for the relationship between them. The new relational table will have attributes corresponding to the keys of the related entities.

Beispiel. Students listen to courses.

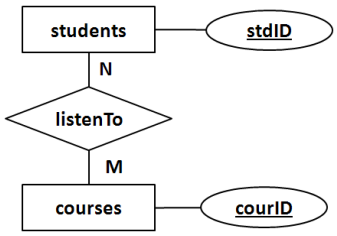


Abbildung 4.6: Example: N:M relationship

students : {[stdID: string; name: string; semester: integer]}
courses : {[courID: string; title: string; semester: string; ...]}
listenTo : {[**stdID**: string; **courID**: string;]}

4.2.3 Generalisation

There are three different possibilities when translating the concept of generalisation from the ER Model to the relational model:

Example. Employees at the University.

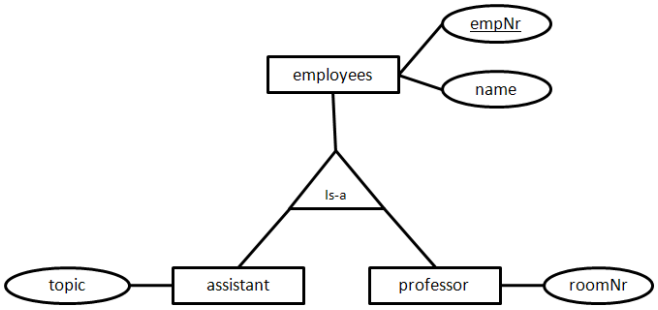


Abbildung 4.7: Example: Generalisation *employees*

- Only the general type:
  employees (empNr, name, topic, roomNr)
  Problem: NULL-values.
- Only the more specific types:
  assistant (empNr, name, topic)
  professor(empNr, name, roomNr)
  Problem: There are employees who are neither assistants nor professors.
- All types:
  employees (empNr, name)
  assistant (empNr, topic)
  professor (empNr, roomNr)
  Problem: Information is spreat across several relations.

4.2.4 Weak-Entities

The primary key of the entity type of which the weak entity type is dependent on will get a part of the primary key of the relational table that illustrates the weak entity type. Consequently, the weak entity has a composite key.

Beispiel. Rooms are situated in houses.

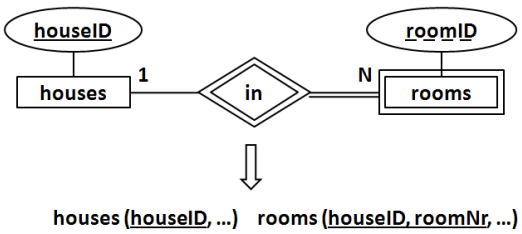


Figure 4.8: Example: Translating the weak entity *rooms*

4.2.5 Simplifying the Schema

After translating relationships there may be relational tables with the same keys. These relational tables can be summarised in one single relation. This may happen in case of 1:N, N:1 and 1:1 relationships because the relation of the relationship has the same key as the relation of the entity.

4.3 Relational Algebra (Formal Relational Query Language)

Beside describing the structure of the relational tables, there exists a language that allows to extract information from the database. There are two formal languages, which have been developed for queries to the database. They are

1. relational algebra (spoken \al-jə-brə\) or formal relational query language (more operational, very useful for representing execution plans) and
2. relational calculus (see mathematics; lets users describe what they want, rather than how to compute it; it is non-operational but declarative)

The relational algebra plays an important role when developing database systems. It is more procedural-structured. It enables a structured plan how a query to the database has to be executed. So, we concentrate on relational algebra in the following.

Basic operators:

- Selection (σ)
- Projection (π)
- Cross-product ($\times$)
- Set-difference ($-$)
- Union ($\cup$)

Additional operators:

- Intersection ($\cap$)
- Join ($\bowtie$, $\ltimes$, $\rtimes$)
- Division ($\div$)
- Renaming: not essential, but useful.

4.3.1 Operations on Sets

The following operations are defined on sets, which base on the same relational schema (that means that these operations are only defined for relational tables having the same attributes).

- Intersection $\cap$ of the sets R and S: $R \cap S = \{t | t \in R \wedge t \in S\}$ (The intersection of the sets R and S equals the set of all t's such that t is the element of R **and** t is the element of S.)
- Union $\cup$ of the sets R and S: $R \cup S = \{t | t \in R \vee t \in S\}$ (The union of the sets R and S equals the set of all t's such that t is the element of R **or** t is the element of S.)
- Set-difference $-$ (or prime is the complement of set S in set R): $R \setminus S = \{t | t \in R \wedge t \notin S\}$ (A set-difference is the complement set of set S in set R equals the set of all t's such that t is the element of R and t is not the element of S.)

r

(A	B	C)
a_1	b_1	c_1
a_1	b_2	c_1
a_2	b_1	c_2

s

(A	B	C)
a_1	b_2	c_1
a_2	b_2	c_1
a_2	b_2	c_2

r ∩ s =

(A	B	C)
a_1	b_2	c_1

r ∪ s =

(A	B	C)
a_1	b_1	c_1
a_1	b_2	c_1
a_2	b_1	c_2
a_2	b_2	c_1
a_2	b_2	c_2

r − s =

(A	B	C)
a_1	b_1	c_1
a_2	b_1	c_2

s − r =

(A	B	C)
a_2	b_2	c_1
a_2	b_2	c_2

Figure 4.9: Some set operations - in general

4.3.2 Selection

The selection σ_φ selects a subset of the tuples of a relation, namely those which satisfy predicate φ. Selections act like a filter on a set. The predicate φ has to contain attributes, which are available in the relation R:

$\sigma_\varphi(R) := \{t | t \epsilon R \wedge fulfillsThePredicate\}$

Beispiel. Have a look at Figure 4.9:

$\sigma_{A=a_1}(r) =?$

$\sigma_{A=a_1}(s) =?$

The selection is commutative with respect to its operator composition:

$\sigma_{A=a_1}(\sigma_{B=b_1}(r)) = \sigma_{B=b_1}(\sigma_{A=a_1}(r))$

A simple selection predicate φ has the form

$< Term >< ComparisonOperator >< Term >.$

$< Term >$ is an expression that can be evaluated to a data value for a given tuple:

- an attribute name,
- a constant value,
- an expression built from attributes, constants, and data type operations such as +, -, *, /.

$< ComparisonOperator >$ is

- $=$ (equals), $\neq$ (not equals),
- $<$ (less than), $>$ (greater than), $\leq$, $\geq$,
- or other data type dependent predicates, e.g. LIKE.

Additionally, it is possible to build well-defined operations on the attribute values, e.g. arithmetic operations on numbers and logical combinations of attributes with the help of ”and” ($\wedge$), ”or” ($\vee$) und negation.

Beispiel. Have a look at Figure 4.9:

$\sigma_{(B=b_1)\wedge\neg(A=a_1)}(s) =?$

4.3.3 Projection

The projection eliminates all attributes (columns) of the input relation but those mentioned in the projection list. It extracts columns (attributes) of the relation R.

$\pi_\beta(R) := \{t_\beta | t \epsilon R\}$

The schema of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.

Beispiel. Have a look at Figure 4.9:

$\pi_{A,B}(r) =?$

A further example **Employees** and **Students**.

Employees	
Name	**Town**
Max Müller	Wien
Tina Schmidt	Salzburg
Klaus Meyer	Wien

$\pi_{Town}(Employees)$

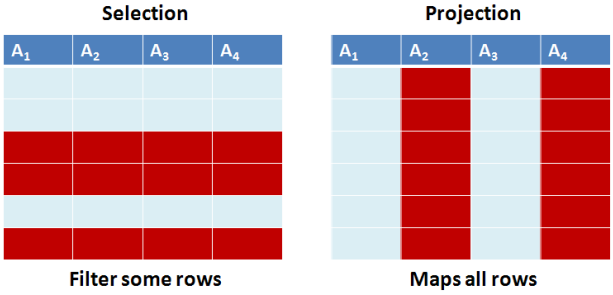
Employees
Town
Wien
Salzburg

Students	
Name	**Town**
Max Müller	Wien
Andre Petersen	Graz
Thomas Ebert	Innsbruck

$\pi_{Name}(Students)$

Students
Name
Max Müller
Andre Petersen
Thomas Ebert

Selection vs. Projection



4.3.4 Cross product or Cartesian product

The cross product allows the combination of each row of the first relation with each row of the second relation. If all attributes are different in the input relations, the result schema contains the sum of all attributes of the both relations. The amount of tuples (rows) in the result schema is the result of the multiplication of the number of rows of the two input relations.

There are two relations $R = (a_1, a_2, ..., a_n)$ and $S = (b_1, b_2, ..., b_m)$. Then, the cross product is defined as follows:

$R \times S := \{(a_1, a_2, ..., a_n, b_1, b_2, ...b_m) | (a_1, a_2, ..., a_n) \epsilon R \wedge (b_1, b_2, ...b_m) \epsilon S\}$

Each row of R is paired with each row of S. The result schema has one field per field of R and S, with field names 'inherited' if possible.

The cross product is only defined in the case that $R \cap S = $. If you want to build the cross-product of two relations that have common attributes, these attributes have to be renamed in one of the two relations.

Example. An example.

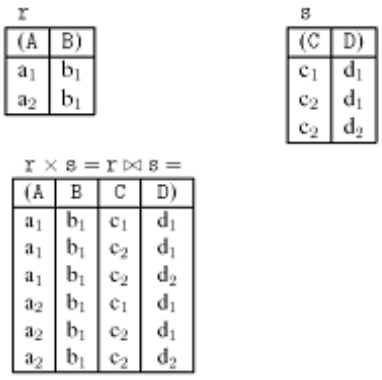


Figure 4.10: Cross product - an example

4.3.5 Join

The intermediate result generated by a Cartesian product may be quite large in general. Since the combination of Cartesian product and selection in queries is common, a special operator join has been introduced.

Condition Join (Theta-Join) The (theta-)join $R \bowtie_\Theta S$ between relations R, S is defined as

$R \bowtie_\Theta S \equiv \sigma_\Theta(R \times S)$.

The join predicate Θ may refer to attribute names of R and S.

Join combines tuples from two relations and acts like a filter: tuples without join partner are removed.

There are join variants, which act like filters only: **left** and **right semi-join** ($\ltimes$, $\rtimes$).

$R \ltimes_\Theta S \equiv \pi_{sch(R)}(R \times_\Theta S)$ or $R \rtimes_\Theta S \equiv \pi_{sch(S)}(R \times_\Theta S)$

or do not filter at all: outer-join (see below).

Natural Join The natural join provides another useful abbreviation. In the natural join $R \bowtie S$, the join predicate Θ is defined to be a conjunctive equality comparison of attributes sharing the same name in R, S.

$R \bowtie S = \pi_{A_1,.....A_m,R.B_1,.....R.B_k,C_1,.....C_n}(\sigma_{R.B_1=S.B_1 \wedge .....\wedge R.B_k=S.B_k}(R \times S))$

Es wird hier also ein Kreuzprodukt gebildet, aus dem dann nur diejenigen Tupel selektiert werden, deren Attributwerte für gleichbenannte Attribute der beiden Argumentrelationen gleich sind. Des weiteren werden diese gleichbenannten Attribute in die Ergebnisrelation nur 1x übernommen. Der Verbundoperator ist kommutativ.

Example. An example.

r

A	B
a_1	b_1
a_2	b_1
a_3	b_2

s

B	C
b_2	c_1
b_2	c_2
b_1	c_3
b_3	c_4

r⋈s

A	B	C
a_1	b_1	c_3
a_2	b_1	c_3
a_3	b_2	c_1
a_3	b_2	c_2

Abbildung 4.11: Natural Join - an example

A further example **Courses**, **Students** and **ListensTo**.

Courses

Short	**Title**
SysInf IV	Datenbanken I
Info III	Softwaretechnologie
Info II	Algorithmen und Datenstrukturen
SysInf I	Digitale Systeme

ListensTo

Name	**Short**
Max Müller	SysInf IV
Max Müller	Info III
Andre Petersen	Info II
Andre Petersen	Info III

Students

Name	**Town**
Max Müller	Wien
Andre Petersen	Graz
Thomas Ebert	Innsbruck

Students ⋈ ListensTo

Name	**Town**	**Short**
Max Müller	Wien	SysInf IV
Max Müller	Wien	Info III
Andre Petersen	Graz	Info II
Andre Petersen	Graz	Info III

Students ⋈ ListensTo ⋈ Courses

Name	**Town**	**Short**	**Title**
Max Müller	Wien	SysInf IV	Datenbanken I
Max Müller	Wien	Info III	Softwaretechnologie
Andre Petersen	Graz	Info II	Algorithmen und Datenstrukturen
Andre Petersen	Graz	Info III	Softwaretechnologie

$\pi_{Name,Title}((\sigma_{Town='Wien'}(\textbf{Students})) \bowtie \textbf{ListensTo} \bowtie \textbf{Courses})$

Name	**Title**
Max Müller	Datenbanken I
Max Müller	Softwaretechnologie

Outer-Join The **join** eliminates tuples without partner:

A	B
a_1	b_1
a_2	b_2

⋈

B	C
b_2	c_2
b_3	c_3

=

A	B	C
a_2	b_2	c_2

The **left outer join** preserves all tuples in its left argument, even if a tuple does not team up with a partner in the join:

A	B
a_1	b_1
a_2	b_2

$\bowtie$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_1	b_1	NULL
a_2	b_2	c_2

The **right outer join** preserves all tuples in its right argument:

A	B
a_1	b_1
a_2	b_2

$\bowtie$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_2	b_2	c_2
NULL	b_3	c_3

The **full outer join** preserves all tuples in both arguments:

A	B
a_1	b_1
a_2	b_2

$\bowtie$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_1	b_1	NULL
a_2	b_2	c_2
NULL	b_3	c_3

4.3.6 Division

The division $\div$ is not supported as a primitive operator, but it is useful for expressing queries such as: "Find sailors who have reserved **all** boats."

Möchte man die Relation r durch s dividieren, so muss die Attributmenge von s eine Teilmenge der Attributmenge von r sein. Das Ergebnis hat die Differenz der Attributmengen als Attribute und wählt jene Tupel aus r aus, die eingeschränkt auf die Differenz der Attribute $R - S$ für alle Tupel aus s denselben Wert haben.

Example. An example.

R	A	B	C	D
	a	b	c	d
	a	b	e	f
	b	c	e	f
	e	d	c	d
	e	d	e	f
	a	b	d	e

S	C	D
	c	d
	e	f

R ÷ S	A	B
	a	b
	e	d

Abbildung 4.12: Division - an example

The division can be described as follows:

$$r \div s = \pi_{R-S}(r) - \pi_{R-S}((\pi_{R-S}(r) \times s) - r)$$

4.3.7 NULL-values

When developing databases it might be necessary to model incomplete data. Imagine, you have to store a person whose address is existent but unknown to you. Besides, it might also be possible that attributes are not useable, for instance, the person has no mobile phone.

Storing incomplete data in a database assumes that the database is able to manage so-called "null-values". In most database management systems these null-values are supported by the keyword **null**.

With respect to relational algebra boolean expressions base on a three-valued logic with the boolean values *true*, *false* and *unknown*. Given the operations *and*, *or* and *not* will deliver following truth tables:

and	*true*	*unknown*	*false*
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

or	*true*	*unknown*	*false*
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

not	
true	false
unknown	unknown
false	true

Example. Assume there are three integer - attributes A, B and C. The value of A is 10, the value of B is 20 and the value of C is null. Then, following results will be delivered:

$A < B$ or $B < C$: true

$A > B$ and $B > C$: false

$A > B$ or $B > C$: unknown

not $B = C$: unknown

5 Design Theory for Relational Databases

Design theory for relational databases enables developing relational schemas with the help of formal methods. It allows fine-tuning of conceptual relational schemas. The basis for this is built by functional dependencies. Functional dependencies are used as generalisation for keys and perform normalisation on relation schemas. Normalisation allows characterising the quality of a relational schema.

5.1 Functional Dependencies

In the following, you will find an abstract relational database schema:

- Schema $\Re = \{A, B, C, D\}$
- Attributes $A, B, C, ...$ Sets of attributes: $\alpha, \beta, ...$
- Relation R, Tuple r, s, t

5.1.1 Definition

A functional dependency (FD) is a constraint on a single instance of a relational database schema. It must hold on every instance of the relation.

Given $\alpha \subseteq \Re$, $\beta \subseteq \Re$. A relation R fulfills a functional dependency $\alpha \rightarrow \beta$ exactly if $\forall r, t \in R$ with $r.\alpha = t.\alpha$, then $r.\beta = t.\beta$.
(dt. Eine Relation R erfüllt eine funktionale Abhängigkeit (FD engl. functional dependency) $\alpha \rightarrow \beta$ genau dann, wenn $\forall r, t \in R$ mit $r.\alpha = t.\alpha$ gilt, $r.\beta = t.\beta$.)

$\alpha \rightarrow \beta$: If two tuples have the same values for each attributes in α, then they have the same values for all attributes in β.

The notation is the following: $\{A_1 A_2 ... A_n\} \rightarrow \{B\}$ or simply $A_1 A_2 ... A_n \rightarrow B$.

The set of attributes $\alpha = \{A_1 A_2 ... A_n\}$ **functionally determine** $\beta = B$. *(dt. "Die $\alpha-$Werte bestimmen die $\beta-$Werte funktional (d.h. eindeutig)." α wird auch als Determinante von β bezeichnet.)*

Example Family with implicit FDs:

Child $\rightarrow$ Father, Mother

Child, Grandma $\rightarrow$ Grandpa

Child, Grandpa $\rightarrow$ Grandma

Child	Mother	Father	Grandma	Grandpa
Sara	Sandra	Martin	Simone	Ludwig
Sara	Sandra	Martin	Lisa	Hubert
Simon	Sandra	Martin	Simone	Ludwig
Simon	Sandra	Martin	Lisa	Hubert
Sara	...	...	Simone	...

Example Schema $\Re = \{A, B, C, D\}$ of relation R

Question: Are the following FDs valid on the relation R?

$\{A\} \rightarrow \{B\}$
$\{C, D\} \rightarrow \{B\}$
$\{B\} \rightarrow \{C\}$
$\{A, B\} \rightarrow \{C\}$
$\{B, C\} \rightarrow \{A\}$
$\{B\} \rightarrow \{A\}$

A	B	C	D
a4	b2	c4	d3
a1	b1	c1	d1
a1	b1	c1	d2
a2	b2	c3	d2
a3	b2	c4	d3

Trivial functional dependencies are FDs, which are automatically fulfilled by each instance of a relation. It is essential: $\alpha \rightarrow \beta$ with $\beta \subseteq \alpha$.

An example for a trivial functional dependency: $A \rightarrow A$.

5.1.2 Keys

Functional dependencies allow us to formally define keys.

Definition Superkey

$\alpha \subseteq \Re$, is superkey, if $\alpha \rightarrow \Re$.

A superkey is a set of attributes that has the uniqueness property but is not necessarily minimal.

Definition Candidate Key or Key

$\alpha \subseteq \Re$ is a candidate key or key, if the following constraints are met:

- Uniqueness: $\alpha \rightarrow \Re$
- Minmality: α is minimal, e.g. $\forall A \in \alpha : (\alpha - \{A\}) \nrightarrow \Re$ (that means that α cannot be minimised any more)

If a relation has mutliple keys, specify one to be the primary key. In a relational schema underline the attributes of the primary key.

5.1.3 Determination of Keys

The task of database designers is to determine functional dependencies with respect to semantics of the modeled mini world.

Example A village is described by its name (Name), federal state (BLand), area code (Vw) and the population (Ew).

Which FDs will be available with respect to real world semantic?

- Village names are unique within a federal state, that means {Name, BLand} → {Vw, Ew}
- Several villages may have the same area code, assuming that they have different names, that means {Name, Vw} → {BLand, Ew}

Which keys can be defined with respect to these FDs?

- {Name, BLand}
- {Name, Vw}

Closure of FDs Having functional dependencies one can derive further dependencies.

Example: Room → PersNr, PersNr → Name ⇒ Room → Name

Given

- a set of attributes $\{A_1A_2...A_n\}$ and
- a set of FDs S,

the closure of $\{A_1A_2...A_n\}$ under the FDs in S is

- the set of attributes $\{B_1B_2...B_m\}$ such that for $1 \leq i \leq m$, the FD $\{A_1A_2...A_n \rightarrow B_i\}$ follows from S.
- the closure is denoted by $\{A_1A_2...A_n\}^+$.

The determination of the closure F^+ of F is realised by Armstrong's Axioms (1974).

Armstrong's Axioms α, β, γ and δ define subsets of attributes of $\Re$. For deriving the closure, the following three aximos are requested:

- Reflexivity (*dt. Reflexivität*): If β is a subset of α, then it is true that $\alpha \rightarrow \beta$. Especially, it is essential that $\alpha \rightarrow \alpha$.
- Augmentation *(dt. Verstärkung)*: If $\alpha \rightarrow \beta$ is true, then it is also true that $\alpha\gamma \rightarrow \beta\gamma$.
- Transitivity *(dt. Transitivität)*: If $\alpha \rightarrow \beta$, $\beta \rightarrow \gamma$ are true, then it is also true that $\alpha \rightarrow \gamma$.

Addtional axioms, which simplify the determination of the closure:

- Union *(dt. Vereinigung)*: If $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ are true, then it is also true that $\alpha \rightarrow \beta\gamma$.
- Decomposition *(dt. Dekomposition)*: If $\alpha \rightarrow \beta\gamma$ is true, then it is also true that $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$. (This allows to define a single attribute on the right side.)
- Pseudo-transitivity *(dt. Pseudotransitivität)*: If $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$ are true, then it is also true that $\gamma\alpha \rightarrow \delta$.

Equivalence of two sets of FDs Sometimes, we are not interested in the whole closure of FDs but only a set of attributes, which are functionally defined by FDs.

Two sets F, G of FDs are equivalent ($F \equiv G$), if they have the same closure, that means that $F^+ = G^+$.

5.2 Normalisation of Database Tables

The normalisation process is fundamental to the prevention of unrequested side effects and redundancy problems in relational schemas. Its purpose is to eliminate ambiguity and inconsistencies in relational schemas caused by data redundancy.

Normalisation is the process of simplifying relational schemas by using a simple set of rules , so-called **normal forms**, to avoid side effects and inconsistences caused by data redundancy in databases.

The **principle of duality** describes the capability of rebuilding the original relations after performed the normalisation process.

Consequentley, relations provide the following advantages:

- No redundant data
- No side effects
- No ambiguity
- No inconsistences

5.2.1 Anomalies and Decomposition

Poorly conceptual design can produce relations hosting non-associated information (that means information not belonging together; unrelated concepts). This relations may cause anomalies in practice.

MOVIE				
MID	TITLE	YEAR	STUDIO	STUDIOADDRESS
1	The Incredibles	2004	Disney	Burbank, CA
2	Bambi	1942	Disney	Burbank, CA
3	Avatar	2010	20th Century	Century City, CA

Abbildung 5.1: Redundant data storage

Anomalies are processing defects provoked by manipulating incorrect relations:

- **Update** anomalies. In case of data changes not all entities will be considered when updating data.
  - When a single mini-world fact needs to be changed, multiple tuples must be updated.
  - Redundant copies get out of sync and it is hard to identify the correct information.
  - Application programs unexpectedly receive multiple answers when a single answer was expected.
- **Insert** anomalies. In case of storing non-associated information in a single relation, new entities can only be entered if non-involved entities or even new relations are at hand.
  - The address of a new studio cannot be inserted until it is known which movies it will produce.
  - Since MID is a key, it cannot be set to NULL.
- **Delete** anomalies. In case of mixing up non-associated information, non-involved entities might get lost when deleting data.
  - When the last movie of a studio is deleted, its address is lost.

Preventing these anomalies may request for decomposing incorrect relations. This process is called **decomposition**. The set of instructions how these relations are decomposed to avoid anomalies is called **normalisation**. The corresponding rules are called **normal forms**. The normal form is a way of measuring the depth to which a database has been normalised. The following normal forms will discussed in more detail: first normal form, second normal form and third normal form.

5.2.2 First Normal Form

Aim. Adjusts non-atomic domains in relations, whereby non-atomic domains are attributes that represent data groups.

Not in 1NF

PARENTS		
FATHER	MOTHER	CHILDREN
Johann	Martha	{Elsa, Lucia}
Johann	Maria	{Theo, Josef}
Heinz	Martha	{Cleo}

In 1NF

PARENTS		
FATHER	MOTHER	CHILD
Johann	Martha	Elsa
Johann	Martha	Lucia
Johann	Maria	Theo
Johann	Maria	Josef
Heinz	Martha	Cleo

Abbildung 5.2: First Normal Form (1NF)

Definition. A relation will be in first normal form (1NF) if each table entry (or attribute value) is atomic (no lists, sets, records, relations).

Transformation into 1NF.

1. Create a new table for each data group.
2. Dissolve non-atomic attributes in the old table.
3. Determine a primary key in each table in case that the original primary key was not clear.

Anamolies before transforming into 2NF.

Update anomaly: If changes are requested in redundant attributes, each tuple that contains redundant data will have to be updated.

Insert anomaly: A new value for an attribute of a composite key can only be inserted when all other key attributes contain values as well. (Null-values are not allowed in primary key attributes!)

Delete anomaly: If the last attribute of a composite key is deleted, the values of the whole tuple will get lost.

You can get rid off these anomalies by implementing the second normal form.

5.2.3 Second Normal Form

Aim. Attributes, which are only dependent on a part of the primary key will be handled.

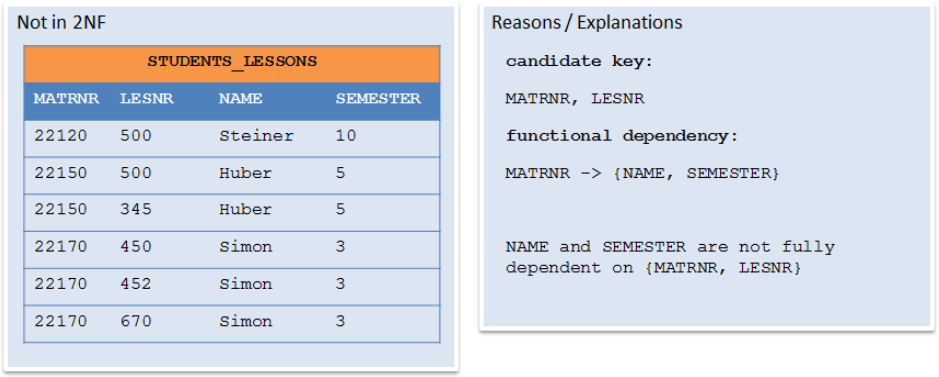


Abbildung 5.3: Second Normal Form (2NF)

Definition. A relation will be in second normal form (2NF) if it is in 1NF and each non-key attribute is fully dependent on the primary key. An attribute is called a non-key attribute (of R), if it does not appear in any minimal key of R. Solution. Decomposition of the original relation into pieces of relations, which all fulfil 2NF:

- LISTENTO: [MATRNR, LESNR]
- STUDENTS: [MATRNR, NAME, SEMESTER]

Shortly spoken, redundances (double attributes) shall be prevented.

Transformation into 2NF.

1. Eliminate attributes (columns), which are not fully dependent on the primary key.
2. Sum up eliminated attributes in a new table, whereby attributes dependent on one key compose a separate table.
3. Insert the corresponding key attributes in the new tables.

Anamolies before transforming into 3NF.

Update anomaly: Transitively dependent domains have to be changed in each tuple that contains values that have to be changed.

Insert anomaly: Transitively dependent values can only be inserted when its relations are inserted before.

Delete anomaly: All dependent values will get lost when deleting the relation.

You can get rid off these anomalies by implementing the third normal form.

5.2.4 Third Normal Form

Aim. Elimination of dependencies among non-key attributes.

Not in 3NF

MUSIC_INTERPRET			
MID	ALBUM	INTERPRET	FOUNDING YEAR
4811	Not That Kind	Anastacia	1999
4823	Freak Of Nature	Anastacia	1999
4712	Wish You Were Here	Pink Floyd	1965

Abbildung 5.4: Third Normal Form (3NF)

Definition. A relation will be in third normal form (3NF) if it is in 2NF and each non-key attribute is not transitively dependent on the primary key. Transitive dependency means that a non-key attribute is dependent on another non-key attribute. Problems:

- FOUNDING_YEAR is dependent on the interpret (NAME) but not from MID.
- FOUNDING_YEAR is stored redundantly.

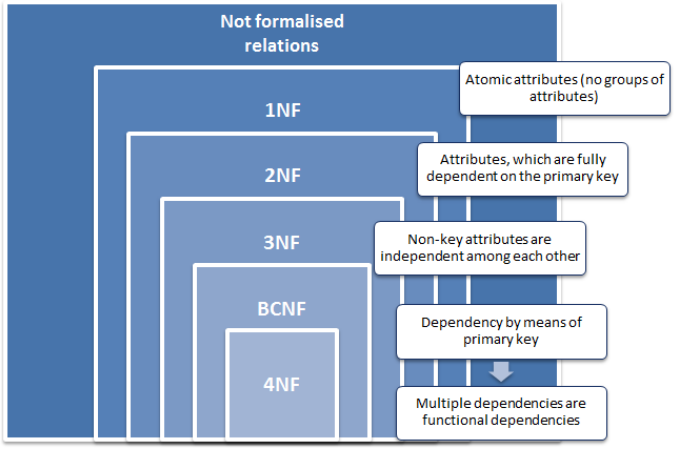
Solution. Modelling as M:N relationship that means introducing new relations:

- MUSIC: [MID, ALBUM]
- INTERPRET: [IID, NAME, FOUNDING_YEAR]
- CD_INTERPRET: [MID, IID]

Transformation into 3NF.

1. Eliminate attributes, which are transitively dependent on the primary key.
2. Create new tables for eliminated attributes, whereby attributes, which are dependent on the same non-key attribute are collected in one table.
3. Add the non-key attribute, which was part of the primary key in the new table.

The process of normalisation should be used for fine adjustment in the conceptual design phase. The ER-Model shall be transformed into relations in 3NF.



5.2.5 Defining keys and determining normal forms

Given a set of functional dependencies for a particular relation it is possible to define keys by using the so-called closure of attributes. Furthermore, we are able to determine the kind of normal form, e.g first, second or third normal form.

Computation of the closure of a set of attributes Given: α a set of attributes and F a set of FDs, written as (F, α)

Requested: attributes, which can be derived by α with the help of F.

Algorithmus AttrClosure

Input: (F, α), Set F of FDs and set of attributes α

Output: Set of attributes α^+

$AttrClosure(F, \alpha)$
$\alpha^+ = \alpha$
while $\exists(\beta \to \gamma) \in F$ with $\beta \subseteq \alpha^+$ and $\gamma \nsubseteq \alpha^+$ do
$\alpha^+ = \alpha \cup \gamma$
return (a^+)

Defining keys with AttrClosure Keys of a relation R can be found with FDs:

(Definition - Repetition) $\alpha \subseteq \Re$ is key, if the following constraints are met:

- Uniqueness: $\alpha \to \Re$
- Minmality: α is minimal, e.g. $\forall A \in \alpha : (\alpha - \{A\}) \nrightarrow \Re$ (that means that α cannot be minimised any more)

Example Defining keys

Given: $\Re = \{A, B, C, D, E, F\}$, $F = \{C \to BDAE\}$

Solution: Heuristic method: all attributes, which are NOT on the right side, might be potential candidates for keys because the cannot be defined with AttrClosure. Consequently, they have to be part of the key.

C and F are not available on the right side, therefore following attempt is started:

$AttrClosure(C \to BDAE, CF) = \{C, F, B, D, A, E\} \Rightarrow CF$ is key of $\Re$.

5.2.6 Examples

1. Computation of the closure of a relation

   Given the relation $\Re = \{A, B, C, D, E, G\}$ there are following eight FDs F given:

   $AB \to C \quad C \to A \quad BC \to D \quad ACD \to B$
   $D \to EG \quad BE \to C \quad CG \to BD \quad CE \to AG$

   It is also essential that: $\alpha = \{BD\}$

   Define the closure of a set of attributes α^+ of (F, α) with the help of AttrClosure.

   (Solution: $\alpha^+ = \{A, B, C, D, E, G\}$

2. There are following FDs F given:

   $F = \{AB \to C, B \to D, CD \to E, CE \to GH, G \to A\}$

   (a) Calculate $(AB)^+$. (Lösung: $(AB)^+ = \{A, B, C, D, E, G, H\}$)
   (b) Is $AB \to E$? (Lösung: ja)
   (c) Is $BG \to C$? (Lösung: ja)

3. Given is the relation $r(R)$:

r	(A	B	C	D	E)
	a1	b1	c1	d1	e1
	a1	b2	c2	d2	e1
	a2	b1	c3	d3	e1
	a2	b1	c4	d3	e1
	a3	b2	c5	d1	e1

Define, which FD in r is valid:

- Is $A \to D$? (Lösung: nein)
- Is $AB \to D$?(Lösung: ja)
- Is $C \to BDE$? (Lösung: ja)
- Is $E \to A$? (Lösung: nein)
- Is $A \to E$? (Lösung: ja)
- Is $A \to BC$? (Lösung: nein)

4. Given is the relation R and the functional dependencies F.

   (a) Define all possible keys for R.

   (b) In which normal form is R? (Explain!)

   i. Example $R = ABCDE$; $F = \{A \to B\}$ein Schlüssel: ACDE;
      - 2NF: nein, da B nicht voll funktional von ACDE abhängig ist;
      - 3NF: nein (ist schon nicht in 2NF)

   ii. Example $R = ABCDEF$; $F = \{A \to B, F \to A, C \to D\}$ein Schlüssel: CEF;
      - 2NF: nein (F->A), da A nicht voll funktional von CEF abhängig;
      - 3NF: nein (ist schon nicht in 2NF)

   iii. Example $R = ABCD$; $F = \{A \to D, B \to A, B \to C, C \to D, A \to B\}$zwei Schlüssel: A; B;
      - 2NF: ja;
      - 3NF: nein, da transitive Abhängigkeit (A->C->D)

   iv. Example $R = ABCDE$; $F = \{A \to BC, E \to D\}$ein Schlüssel: AE;
      - 2NF: nein, da weder BC noch D voll funktional abhängig von AE;
      - 3NF: nein

   v. Example $R = ABCD$; $F = \{AB \to C, C \to D, D \to AB\}$drei Schlüssel: AB; C; D;
      - 2NF: ja;
      - 3NF: ja (es gibt kein nicht-primitives Attribut, d.h. alle sind Schlüsselattribute)

   vi. Example $R = ABC$; $F = \{B \to AC\}$ein Schlüssel: B;
      - 2NF: ja;
      - 3NF: ja

5. For the relation $R = \{A, B, C, D, E, F\}$ there are following functional dependencies given: $F = \{A \to D, B \to C, B \to A, D \to F, D \to E, E \to D\}$

   Define all candidate keys for R. How can you use the closure of attributes?

   Lösung
   $\{A \to D\}$ $A^+ = \{A, D, E, F\}$
   $\{B \to C\}$ $B^+ = \{A, B, C, D, E, F\}$
   $\{B \to A\}$ $C^+ = \{C\}$
   $\{D \to F\}$ $D^+ = \{D, E, F\}$
   $\{D \to E\}$ $E^+ = \{D, E, F\}$
   $\{E \to D\}$ $F^+ = \{F\}$

   $B^+ = \{A, B, C, D, E, F\}$

6. For the relation $R = \{A, B, C, D\}$ there are following functional dependencies given: $F = \{AB \rightarrow C,\ B \rightarrow D\}$

   In which normal form is this relation?

   Lösung:

   (a) Schlüsselkandidaten: (AB)
   (b) Nichtschlüsselattribute: (C und D)
   (c) C ist voll funktional abhängig von AB
   (d) D ist NICHT voll funktional abhängig von AB (wegen $B \rightarrow D$)
   (e) nicht in 2NF

7. For the relation$R = \{A, B, C, D\}$ there are following functional dependencies given: $F = \{AB \rightarrow C,\ C \rightarrow D,\ D \rightarrow A\}$

   In which normal form is this relation?

   Lösung:

   (a) Schlüsselkandidaten: (AB; BC; BD)
   (b) Nichtschlüsselattribute: keine vorhanden
   (c) in 2NF, da keine Nichtschlüsselattribute vorhanden
   (d) in 3NF, da für jede FD $\alpha \rightarrow B$ muss mindestens eine der drei Bedingung gelten:
      i. α ist Superschlüssel
      ii. B ist prim (Schlüsselattribut)
      iii. $\alpha \rightarrow B$ ist trivial (d.h. $B \in \alpha$)
   (e) Schlüsselkandidaten sind: AB; BC; BD
      i. das bedeutet für $AB \rightarrow C$
         A. AB ist Superschlüssel
         B. C ist prim
         C. $AB \rightarrow C$ ist trivial
      ii. das bedeutet für $C \rightarrow D$
         A. C ist Superschlüssel
         B. D ist prim
         C. $C \rightarrow D$ ist trivial
      iii. das bedeutet für $D \rightarrow A$
         A. D ist Superschlüssel
         B. A ist prim
         C. $D \rightarrow A$ ist trivial

      R befindet sich in 3NF
   (f) in BCNF, da für jede FD $\alpha \rightarrow B$ muss mindestens eine der zwei Bedingung gelten:
      i. α ist Superschlüssel
      ii. $\alpha \rightarrow B$ ist trivial (d.h. $B \in \alpha$)

5.3 Minimal Basis *(dt. Kanonische (oder minimale) Überdeckung)*

To compute a possible mimial set of FDs, the minimal basis will be defined. The minimal basis allows checking consistency constraints in case of data modifications.

F_C is called **minimal basis** of a set of FDs F, if the following criteria are met:

- $F_C^+ = F^+$ (F_C is equivalent to F)
- In F_C exist no FDs, which have redundant attributes.
- Each left side of a FD in F_C is unique.

A minimal basis F_C exists to each set of functional dependencies F.

Computation of the minimal basis The computation of the minimal basis is derived from its definition:

1. Decompose all FDs with decomposition on the right side.
2. Reduce redundant attributes as follows:
   (a) Perform for each FD $\alpha \to B \in F$ the left reduction, *(dt. kann aus α ein Attribut gekürzt werden, so dass das Ergebnis äquivalent zur ursprünglichen Menge von FDs bleibt?)*
   $\forall A \in \alpha$: is it true that $B \subseteq AttrClosure(F, \alpha - A)$ (is A redundant?)
   if yes, then exchange $\alpha \to B$ by $(\alpha - A) \to B$
   (b) Perform for each (remaining) FD $\alpha \to B \in F$ the right reduction, *(dt. kann B gekürzt werden, so dass das Ergebnis äquivalent zur ursprünglichen Menge von FDs bleibt?)*
   is it true that $B \subseteq AttrClosure(F - (\alpha \to B), \alpha)$ (is B or $\alpha \to B$ redundant?)
   if yes, then eliminate $\alpha \to B$
3. Compose FDs with the help of union axiom.

Example Minimal Basis

$F = \{A \to B,\ B \to C,\ AB \to C\}$

1. Decomposition is not necessary because all attributes on the right side are already single attributes.
2. Reduction
   (a) Left Reduction (Basis: $F = \{A \to B,\ B \to C,\ AB \to C\}$):
   $A \to B$ is already reduced
   $B \to C$ is already reduced
   $AB \to C$:
      i. B redundant? $C \in AttrClosure(F, A) = \{A, B, C\}$ Yes, eliminate $B \Rightarrow F = \{A \to B,\ B \to C,\ A \to C\}$
      ii. A redundant? $AttrClosure(F, B) = \{B\}$ - Check not necessary
   (b) Right Reduction (Basis: $F = \{A \to B,\ B \to C,\ A \to C\}$):
   $A \to B$:
      i. is B redundant? $B \in AttrClosure(F - (A \to B),\ A) = \{A, C\}$ No
   $B \to C$:
      i. is C redundant? $C \in AttrClosure(F - (B \to C),\ B) = \{B\}$ No
   $A \to C$:
      i. is C redundant? $C \in AttrClosure(F - (A \to C),\ A) = \{A, B, C\}$ Yes, eliminate $A \to C \Rightarrow F = \{A \to B,\ B \to C\}$
3. Union axiom cannot be executed $\Rightarrow F_C = \{A \to B,\ B \to C\}$

5.3.1 Examples

1. Computation of the minimal basis
   $F = \{A \to BD,\ AC \to E,\ CD \to E,\ E \to A,\ D \to C\}$
   (Solution: $F_C = \{A \to BD,\ D \to EC,\ E \to A\}$)
   (a) Dekomposition:
   $F = \{A \to B,\ A \to D,\ AC \to E,\ CD \to E,\ E \to A,\ D \to C\}$
   (b) Kürzen
      i. Linksreduktion
      $A \to B$ ist ok
      $A \to D$ ist ok
      $E \to A$ ist ok
      $D \to C$ ist ok
      $AC \to E$: $E \in AttrHülle(F, A)$ ja, d.h. C ist überflüssig $\Rightarrow F = \{A \to B,\ A \to D,\ A \to E,\ CD \to E,\ E \to A,\ D \to C\}$
      $CD \to E$: $E \in AttrHülle(F, C)$ nein; $E \in AttrHülle(F, D)$ ja, d.h. C ist überflüssig $\Rightarrow F = \{A \to B,\ A \to D,\ A \to E,\ D \to E,\ E \to A,\ D \to C\}$
      ii. Rechtsreduktion
      $A \to B$: $B \in AttrHülle(F - (A \to B),\ A)$ nein
      $A \to D$: $D \in AttrHülle(F - (A \to D),\ A)$ nein
      $A \to E$: $E \in AttrHülle(F - (A \to E),\ A)$ ja, d.h. $A \to E$ ist überflüssig $\Rightarrow F = \{A \to B,\ A \to D,\ D \to E,\ E \to A,\ D \to C\}$
      $D \to E$: $E \in AttrHülle(F - (D \to E),\ D)$ nein
      $E \to A$: $A \in AttrHülle(F - (E \to A),\ E)$ nein
      $D \to C$: $C \in AttrHülle(F - (D \to C),\ D)$ nein
   (c) Zusammenfassen: $F_C = \{A \to BD,\ D \to EC,\ E \to A\}$

5.3.2 More Examples

1. There are following FDs F given:
   $F = \{AB \to C,\ DE \to AB,\ A \to D,\ B \to E,\ DE \to C,\ AD \to G\}$
   - Calculate $(AB)^+$. (Lösung: $(AB)^+ = \{ABCDEG\}$)
   - Calculate$(DE)^+$. (Lösung $(AB)^+ = \{DEABCG\}$)
   - Is $AB \equiv DE$? (Lösung: ja - Hüllen sind gleich)
   - Is $AB \to CDG$? (Lösung: ja)
   - Is $DE \to C$ in F redundant? (Lösung: ja)
   - Is D in $AD \to G$ redundant ? (Lösung: ja)
   - Calculate the minimal basis of F.
     (Lösung: $F = \{DE \to A,\ DE \to B,\ A \to D,\ B \to E,\ DE \to C,\ A \to G$ oder $F = \{AB \to C,\ DE \to A,\ DE \to B,\ A \to D,\ B \to E,\ A \to G\}$)
2. Calculate the minimal basis.
   $F = \{AB \to CD,\ B \to D,\ CD \to A,\ C \to E,\ EC \to AB\}$
   (Lösung: $F = \{AB \to C,\ B \to D,\ C \to AEB\}$)
3. Given the relation $\Re = \{A, B, C, D, E, F\}$ there are following eight FDs F given:
   - $A \to BC$
   - $C \to DA$
   - $E \to ABC$
   - $F \to CD$

- $CD \rightarrow BEF$

(a) Calculate the minimal basis.
(Lösung: $F_C = \{A \rightarrow C,\ E \rightarrow A,\ F \rightarrow CD,\ C \rightarrow BEF\}$)
Dekomposition:
$F = \{A \rightarrow B,\ A \rightarrow C,\ C \rightarrow D,\ C \rightarrow A,\ E \rightarrow A,\ E \rightarrow B,\ E \rightarrow C,\ F \rightarrow C,\ F \rightarrow D,\ CD \rightarrow B,\ CD \rightarrow E,\ CD \rightarrow F\}$

i. Kürzen

A. Linksreduktion

- $CD \rightarrow BEF$
  ist C überflüssig? $BEF \in AttrHülle(F, D) = \{D\}$ nein
  ist D überflüssig? $BEF \in AttrHülle(F, C) = \{DABCBEF\}$ ja, d.h. $C \rightarrow BEF$
  $\Rightarrow F = \{A \rightarrow B,\ A \rightarrow C,\ C \rightarrow D,\ C \rightarrow A,\ E \rightarrow A,\ E \rightarrow B,\ E \rightarrow C,\ F \rightarrow C,\ F \rightarrow D,\ C \rightarrow B,\ C \rightarrow E,\ C \rightarrow F\}$

B. Rechtsreduktion

- $A \rightarrow B$: $B \in AttrHülle(F - (A \rightarrow B),\ A) = \{ACDBEF\}$ ja, d.h. $A \rightarrow B$ ist überflüssig
  $\Rightarrow F = \{A \rightarrow C,\ C \rightarrow D,\ C \rightarrow A,\ E \rightarrow A,\ E \rightarrow B,\ E \rightarrow C,\ F \rightarrow C,\ F \rightarrow D,\ C \rightarrow B,\ C \rightarrow E,\ C \rightarrow F\}$
- $A \rightarrow C$: $C \in AttrHülle(F - (A \rightarrow C),\ A) = \{A\}$ nein
- $C \rightarrow D$: $D \in AttrHülle(F - (C \rightarrow D),\ C) = \{CABEFAD\}$ ja, d.h. $C \rightarrow D$ ist überflüssig
  $\Rightarrow F = \{A \rightarrow C,\ C \rightarrow A,\ E \rightarrow A,\ E \rightarrow B,\ E \rightarrow C,\ F \rightarrow C,\ F \rightarrow D,\ C \rightarrow B,\ C \rightarrow E,\ C \rightarrow F\}$
- $C \rightarrow A$: $A \in AttrHülle(F - (C \rightarrow A),\ C) = \{CDBEFA\}$ ja, d.h. $C \rightarrow A$ ist überflüssig
  $\Rightarrow F = \{A \rightarrow C,\ E \rightarrow A,\ E \rightarrow B,\ E \rightarrow C,\ F \rightarrow C,\ F \rightarrow D,\ C \rightarrow B,\ C \rightarrow E,\ C \rightarrow F\}$
- $E \rightarrow A$: $A \in AttrHülle(F - (E \rightarrow A),\ E) = \{EBCF\}$ nein
- $E \rightarrow B$: $B \in AttrHülle(F - (E \rightarrow B),\ E) = \{EACBEF\}$ ja, d.h. $E \rightarrow B$ ist überflüssig
  $\Rightarrow F = \{A \rightarrow C,\ E \rightarrow A,\ E \rightarrow C,\ F \rightarrow C,\ F \rightarrow D,\ C \rightarrow B,\ C \rightarrow E,\ C \rightarrow F\}$
- $E \rightarrow C$: $C \in AttrHülle(F - (E \rightarrow C),\ E) = \{EAC\}$ ja, d.h. $E \rightarrow C$ ist überflüssig
  $\Rightarrow F = \{A \rightarrow C,\ E \rightarrow A,\ F \rightarrow C,\ F \rightarrow D,\ C \rightarrow B,\ C \rightarrow E,\ C \rightarrow F\}$
- $F \rightarrow C$: $C \in AttrHülle(F - (F \rightarrow C),\ F) = \{FD\}$ nein
- $F \rightarrow D$: $D \in AttrHülle(F - (F \rightarrow D),\ F) = \{FCBEF\}$ nein
- $C \rightarrow B$: $B \in AttrHülle(F - (C \rightarrow B),\ C) = \{CEFA\}$ nein
- $C \rightarrow E$: $E \in AttrHülle(F - (C \rightarrow E),\ C) = \{CBFD\}$ nein
- $C \rightarrow F$: $F \in AttrHülle(F - (C \rightarrow F),\ C) = \{CBEA\}$ nein

ii. Zusammenfassen: $F_C = \{A \rightarrow C,\ E \rightarrow A,\ F \rightarrow CD,\ C \rightarrow BEF\}$

(b) Calculate the closure of A. (Lösung: $A^+ = \{A, B, C, D, E, F\}$)

(c) Define all candidate keys. (Lösung: $\{A\}, \{C\}, \{E\}, \{F\}$)

4. Überprüfen Sie, ob die folgenden vier FD-Mengen alle äquivalent sind.

$F_1 = \{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow C\}$

$F_2 = \{A \rightarrow B,\ B \rightarrow C,\ AB \rightarrow C\}$

$F_3 = \{A \rightarrow B,\ B \rightarrow C\}$

$F_4 = \{A \rightarrow B,\ B \rightarrow BC,\ AC \rightarrow BC\}$

(Lösung: FD-Mengen sind äquivalent, da gilt: zwei FD-Mengen sind äquivalen, wenn ihre Hüllen identisch sind: $F_1^+ = \{ABC\} \equiv F_2^+ = \{ABC\} \equiv F_3^+ = \{ABC\} \equiv F_4^+ = \{ABC\}$)

6 The Database Language SQL

Structured Query Language

7 Key Terms [3]

Associative table. A database table that stores the valid combinations of rows from two other tables. An associative table resolves a many-to-many relationship.

Column. The component of a database table that contains all of the data of the same name and type across all rows.

Composite key

Data integrity

Data modeling. A process of defining the entities, attributes and relationships between the entities in preparation for creating the physical database.

Data normalisation

Decomposition process

Defining association

Determinant attribute

Exception conditions

Field. The smallest piece of information that can be retrieved by the database query language. A field is found at the intersection of a row and a column in a database table.

File system. A collection of individual files accessed by application programs.

First normal form

Foreign key. A column (or columns) in a table that draws its values from a primary or unique key column in another table. A foreign key assists in ensuring the data integrity of a table.

Functional dependency

Joining

Many-to-many relationship. A relationship type between tables in a relational databse where one row of a given table may be related to many rows of another table, and vice versa. Many-to-many relationships are often resolved with an intermediate associative table.

Non-loss decomposition

Normal forms

Null value

Object-relational database. A relational database that includes additional operations and components to support object-oriented data structures and methods.

One-to-many relationship. A relationship type between tables where one row in a given table is related to many other rows in a child table. The reverse condition, however, is not true. A given row in a child table is related to only one row in the parent table.

One-to-one relationship. A relationship type between tables where one row in a given table is related to only one or zero rows in a second table. This relationship type is often used for subtyping.

Partial functional dependency

Primary key. A column (or columns) in a table that makes the row in the table distinguishable from every other row in the same table.

Referential integrity. A method employed by a relational database system that enforces one-to-many relationships between tables.

Relation. A two-dimensional structure used to hold related information, also known as table.

Relational database. A collection of tables that stores data without any assumptions as to how the data is related within the tables or between the tables.

Relational integrity

Row. A group of one or more data elements in a database that describes a person, place, or thing.

Second normal form

Table. The basic construct of a relational database that contains rows and columns of related data.

Third normal form

Transitive dependency

An entity is an object that exists and that is distinguishable from other objects.

An attribute is a property or characteristic of an entity type that is of interest to an organization.

A relationship is an association among several entities.

intersection	Schnittpunkt, Durchschnitt
subtype	Untertyp (is-a)
intermediate	dazwischen, Zwischen...
on the spur of the moment	spontan, auf Anhieb, ad hoc
to depict sth.	darstellen, beschreiben

References

[1] Peter Rob Carlos Coronel, Steven Morris. *Database Systems: Design, Implementation, and Management (with Premium WebSite Printed Access Card and Essential Textbook Resources Printed Access Card)*. Course Technology; 10 edition, 2012.

[2] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[3] Mark L. Gillenson. *Introduction to Database Management*. Wiley Pathways, 2008.

[4] Shamkant B. Navathe. Evolution of data modeling for databases, 2012.