Introduction to Information Systems

Course Summary

B Pijnacker | s4106164

Contents

About this summary

This summary follows the same chapter/section/subsection numbering and titles as the book *Fundamentals of Database Systems Seventh Edition Global Edition*. Any section consisting solely of examples are not included in the summary and small examples within section also have not been included.

Part I

Introduction to Databases

Chapter 1

Databases and Database Users

A general database system you might think of when you hear 'database', like searching through a library catalog, is called a **traditional database application**. In these applications, data is mostly textual or numeric. In more modern applications, there is also the need to store more nontraditional data such as images and videos. New types of databases have been created for this, often called **big data storage**.

1.1 Introduction

A database is a collection of related facts that can be recorded and that have implicit meaning. A database has the following implicit properties:

- A database represents some smll subset of the real world, called the universe of discourse;
- A database is a logically coherent collection of data with some inherent meaning;
- A database is designed, built, and populated with data for a specific purpose

A database management system (DBMS) is a computerized system that enables users to create and maintain a database. This involves defining, constructing, and manipulating the database. **Defining** a database means to specify datatypes, structures, and constraints necessary for data to be stored. **Constructing** is the process of storing the data. **Maintaining** the database is the process of querying data in an existing database. We call the combination of a database and a DBMS a database system.

1.3 Characteristics of the Database Approach

There exist a number of characteristics of the database approach that seperate it from more traditional ways of storing data. We describe each of these in a seperate subsection of this section.

1.3.1 Self-Describing Nature of a Database System

A database system contains not only the database itself, but also the database structure and constraints. This means a database system contains all the information you might need about the database that is included. This information is called meta-data. In more traditional file processing, data definition is often part of the application program, and are thus constrained to only one specific database. This is ont the case with a DBMS, as this can extract database definitions from the catalog.

1.3.2 Insulation between Programs and Data, and Data Abstraction

Programs access data through the DBMS. This means that a program is not concerned about the internal workings of the database system. This allows for changes to the interal workings of the database, without needing to rewrite any program that accesses the database. This concept is called data abstraction.

1.3.3 Support of Multiple Views of the Data

A DBMS can support creating multiple views of data, where only relevant data is given to any user, in a format that they require that data to be in. The data might not be stored exactly in this way, but the DBMS can translate it for the user easily.

1.3.4 Sharing of Data and Multiuser Transaction Processing

If multiple people access the database system at the same time, the DBMS can use transaction processing to handle concurrent queries. This makes sure no two people can overwrite the same data simultaniously.

1.4 Actors on the Scene

There exist a lot of different jobs to do with database management and use, in this section we will go through the jobs that involve the database contents. The following section highlights the jobs behind the scenes.

1.4.1 Database Administrators

The database adminstrator is resposible for authorizing access to the database, coordinating its use, and acquiring software and hardware resources when needed. The DBA is accountable for security and system response time.

1.4.2 Database Designers

Database designers are responsible for identifing the data to be stored in the databae and for choosing appropriate structures to represent and store this data. Database designers often are most in use before a database system is deployed.

1.4.3 End Users

End users are the people who need access to the database. The database primarily exists for their use. There are several catagories of end users.

- **Casual end users** These people need occasional access to the database, but often with different information needs every time.
- **Naive end users** Their main job consists of updating data into the database or querying data from the database using standard queries and the same data every time.
- **Sophisticated end users** These are people who thoroughly familiarize themselves with the facilities of the DBMS and implement functions into their own applications.
- **Standalone users** There are people that maintain personal databases that provide easy to use interfaces for them.

1.4.4 System Analysts and Application Programmers (Software Engineers)

System analists determine the requirements of end users and dvelop specifications to meet these requirements. Application programmers then implement these specifications in applications that end users will utilize.

1.5 Workers behind the Scene

- **DBMS system designers and implementers** design and implement the DBMS modules and interfaces in a software package.
- **Tool developers** design and implement tools for use with a DBMS.
- **Operators and maintenance personnel** are responsible for actually running the database hardware and software.

1.6 Advantages of Using the DBMS Approach

1.6.1 Controlling Redundancy

Redundancy means storing the same data multiple times for multiple users. Using a DBMS removes this redundancy, because it stores the data once, and then uses processing to present this data in a way that the user wants.

1.6.2 Restricting Unauthorized Access

A DBMS can provide a security and authorization subsystem, so only users with the correct credentials can access some data and change certain features of the database.

1.6.3 Providing Persistent Storage for Program Objects

A database that stores objects made by class definitions in an OOP language can be persistent storage. This means that after program execution, the data remains stored, which another (or the same) program can retrieve later.

1.6.4 Providing Storage Structures and Search Techniques for Efficient Query Processing

The query processing and optimization module of the DBMS is responsible for choosing an efficient query execution plan for each query. This allows very efficient query execution and thus low overhead for the system and fast execution times for users.

1.6.5 Proving Backup and Recovery

A backup and recovery subsystem of the DBMS can make backups of the database to restore later on when, for example, a disk failure occurs.

1.6.6 Providing Multiple User Interfaces

A DBMS supports a wide variety of user interfaces to access the database. Any of such interfaces is made for a different use case.

1.6.7 Representing Complex Relationships among Data

A database can include numerous varieties of data that are interrelated in many ways. DBMSs must have the capability to represent these complex relationships among the data.

1.6.8 Enforcing Integrity Constraints

Databases have constraints to which all data must hold. If any of these constraints are broken, the DBMS must either handle this or inform somebody that can handle the issue. For example, any relation must have a unique value, also called a key. If a key in a new tuple is equal to one that already exists, a constraint is broken.

1.6.9 Permitting Inferencing and Actions Using Rules and Triggers

A DBMS must allow rules and triggers to exist. These are triggers consist of a condition and a procedure. If the condition is evaluated as true, the procedure is executed.

1.6.10 Additional Implications of Using the Database Approach

More implications include

- Potential for Enforcing Standards;
- Reduced Application Development Time;
- Flexibility;
- Availablity of Up-to-Date Information;
- Economies of Scale.

1.7 A Brief History of Database Applications

1.7.1 Early Database Applications Using Hierarchical and Network Systems

In early systems, a big problem was the intermixing of conceptual relationships with the physical storage and placement of records on disk. These systems did not provide data abstraction as is possible in a modern DBMS.

1.7.2 Providing Data Abstraction and Application Flexibility with Relational Databases

Relational databases were originally proposed to save disk space for data and to provide a mathematical foundation for querying and data representation. Early relational systems were quite slow, since the storage techniques used were very much not optimal.

1.7.3 Object-Oriented Applications and the Need for More Complex Databases

With the emerge of object-oriented programming languages also came the need of object-oriented databases. Initially, they were considered a competitor to relational databases, but they had limited early-on use. Currently, relational DBMSs now include object-oriented concepts, and OODBs are not widely used.

1.7.4 Interchanging Data on the Web for E-Commerce Using XML

With the rapid spread of the world wide web came the need to store websites efficiently and provide easy ways to travel between these with hyperlinks. For this reason came HTML. A way of interchanging data between a database and web pages is XML.

1.7.5 Extending Database Capabilities for New Applications

Basic relational database systems were not capable enough for larger and more complex tasks like storing images or doing data mining. This led to DBMS developers adding a lot of functionality to their systems to incorporate these features.

1.7.6 Emergence of Big Data Storage Systems and NOSQL Databases

New types of database systems were necessary to store the huge amounts of data collected on the internet. For this reason, NOSQL databases came to life. These are databases where some of the data is stored using SQL, but other data is stored without using SQL because some requirements weren't compatible with SQL standards.

1.8 When Not to Use a DBMS

A DBMS has quite a high overhead cost compared to alternatives due to the following:

- High initial investment in hardware, software, and training;
- Overhead for providing security, concurrency control, recovery, and integrity functions.

It may be more desirable to develop custom database applications under the following circumstances:

- Simple, well-defined database applications that are not expected to change at all;
- Stringent, real-time requirements that cannot be met with DBMS overhead;
- Embedded systems with limited storage capacity, where a DBMS would not fit;
- No mulitple-user access to data.

Chapter 2

Database System Concepts and Architecture

2.1 Data Models, Schemas, and Instances

A data model is used to describe the structure of a database. It provides the necessary means to obtain data abstraction.By structure of the database, we mean the data types, relationships, and constraints that apply. Often, data models also include a set of basic operations for retrievals and updates to the database.

2.1.1 Catagories of Data Models

Conceptual Data Models use concepts such as entities, attributes, and relationships. An example of this is the ER-model. This is a high-level data model as it provides concepts close to the way users percieve data.

Representional or Implementation Data Models are most frequently used in commercial DBMSs. They include relational data models, the network, and hierarchical models. Representational data models represent record structures and hence are often called record-based data models.

Physicial Data Models describe how data is stored by the computer, and describes how it should represent data.

Self-describing Data Models are models that describe the data and specifies the data values in the same place. A traditional DBMS does not use this model. These models include XML and NOSQL systems.

2.1.2 Schemas, Instances, and Database State

The description of a database is called the database schema. It is specified during database design and not expected to change frequently. Each object in a schema (such as STUDENT, or COURSE) is caled a schema construct. A schema diagram does not display data such as datatypes or relationships.

A database in any point of time is in a database state, also called the current set of instances. The state of a database includes any data stored, because changing data changes the state. If a database has a defined schema, but no data, its state is called an empty state. The DBMS is responsible for ensuring the database is always in a valid state; that no constraints are being violated.

2.2 Three-Schema Architecture and Data Independence

2.2.1 The Three-Schema Architecture

The goal of the three-schema architecture is to seperate user applications from the physical database. In this architecture, we define different schemas for three levels:

1. The internal level has a schema that describes the physical implementation of the database. It uses a physical data model.

2. The conceptual level has a conceptual schema which describes the structure of the database. It hides the details of physical implementation and concentrates on describing the database using entities, relationships, attributes, data types, constraints, etc. Usually, a representational data model is used.

3. The external level includes a number of user views. These can differ per user or target group. It hides data the user doesn't need and only shows the user what they're interested in.

2.2.2 Data Independence

The three-schema architecture benefits from data independence, the capacity to change the schema at one level without having to change the schema at other levels. There exist two types of data independence. Logical data independence allows us to change the conceptual schema without having to change implementation details, and physical data independence allows us to change the implementation details without having to change the conceptual schema or user views. Physical data independence generally exists in most databases.

2.3 Database Languages and Interfaces

2.3.1 DBMS Languages

The first step after designing a database and choosing a DBMS is specifying the schemas. In many DBMSs, this is done using a data definition language, that is used by the database administrators. Another language, the storage definition language, is used to specify the internal schema of the database. For a true three-schema architecture, we would also need a view definition language. In relational databases, SQL is used for this.

Once the database schemas are compiled and the database is populated, users must have some means to manipulate the database. For this exists the data manipulating language that includes all the operations necessary for retrieval, insertion, deletion, and modification of data.

In modern DBMSs, these languages are not considered distinct languages; rather, one language is used that includes all these features.

2.4 The Database System Environment

2.4.1 DBMS Component Modules

This entire subsection references from a figure in the book on page 73. I suggest you just read that.

2.4.2 Database System Utilities

In addition the the component modules, most DBMSs have certain utilities to help the DBA manage the system. Common utilities have the following types of functions:

- **Loading**. A loading utility is used to load existing files into the database.
- **Backup**. A backup utility creates a backup copy of the database that can be restored must the need arise.
- **Database storage reoganization**. This utility can be used to reorganize the a set of files into different file organizations to improve performance.
- **Performance monitoring**. Allows for monitoring the performance of the database for decision making by the DBA.

2.4.3 Tools, Application Environments, and Communications Facilities

Other tools are often available to database designers, users, and the DBMS. A **database dictionary** might store catalog information, design decisions, usage standards, and user information. **Application development environments** provide an environment for developing database applications.

2.5 Centralized and Client/Server Architectures for DBMSs

2.5.1 Centralized DBMSs Architecture

In older systems, users accessed the database through a terminal that did not have computing power, and only provided a display to a mainframe session. Because of this, all computing that had to be done was done on one centralized DBMS system, that users could connect to.

2.5.2 Basic Client/Server Architectures

When users begun to replace their terminal by a PC, a DBMS could make user of local computing power of the user. The idea is to define specialized servers with specific functionalities. This include file servers, mail servers, and web servers. The client machine provides the user with the interface to utilize these servers.

2.5.3 Two-Tier Client/Server Architectures for DBMSs

If a client wished to access the DBMS, the program establishes a connection to the DBMS, through which the client can communicate with the DBMS as long as both the client and server have the right software installed. This is called two-tier, because the software is distributed over server and client.

2.5.4 Three-Tier and n-Tier Architectures for Web Applications

The three-tier architecture adds an intermediate layer between the client and the database server. This is called the application server, or web server, depending on the use case. This server might check for credentials of the user, so that a user cannot access data that they are not supposed to.

2.6 Classification of Database Management Systems

Several criteria can be used to classify DBMSs. The first is the data model in which it is based. For most current commercial DBMSs, this is the relational data model, of which the systems are known as SQL systems. Some experimental DBMSs are based on the XML model, which is tree structred. For more information about these models, read the last page and a half of this section in the book (page 82, 83).

A second criterion is the number of users supported. This makes the distinction between single-user and multi-user systems.

A third criterion is the number sites over which the database is distributed. A DBMS can be centralized on a single computer, or distributed with multiple computers over a network. Homogeneous DMBSs use the same software at all sites, which heterogeneous DBMSs don't. They might use different software for every site if this is needed.

The fourth criterion is cost. There exist free DBMSs, but also systems that might cost millions of dollars for large database systems. Which one to choose here is mostly up to what you can afford (and want to pay) and what special functionality you might need from a DBMS that others can't offer.

Part II

Conceptual Data Modeling and Database Design

Chapter 3

Data Modeling Using the Entity-Relationship (ER) Model

3.1 Using High-Lvel Conceptual Data Models for Database Design

Designing a database consists of a few distinct steps. The first of these is **requirements collection and analysis**. In this step the database designers interview users to collect their requirements. The result of this step is a concisely written set of user requirements. The next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. These two steps are DBMS independent.

Next follows the implementation of the database in a commercial DBMS. This consists of application program design, physical design, and transaction implementation. In these steps any necessary schemas are created. How this happens depends on the DBMS used.

3.3 Entity Types, Entity Sets, Attributes, and Keys

3.3.1 Entities and Attributes

The basic ER model represents entities which have attributes. An entity represents a *thing* or *object* in the real world. An entity has attributes; the physical properties that describe it.

An attribute can be composite or atomic. An atomic attribute is one that holds a single value, and a composite attribute is one that holds multiple attributes within it. An attribute can also be single- or multivalued. This being different from atomic and composite, because multivalued is still a singular attribute, just with multiple values as opposed to an attribute with multiple distinct attributes within it.

A particular attribute may also contain a NULL value. This means that this attribute might be unknown, nonexistent, or not applicable for a certain entity.

3.3.2 Entity Types, Entity Sets, Keys, and Value Sets

Entity Types and Entity Sets

A database usually contains groups of entities that are similar. An entity type defines a set of entities that have the same attribute. An example of this is an entity type EMPLOYEE or COMPANY.

Key Attributes of an Entity Type

An important property of an entity type is the key, or uniqueness constraint. A entity collection cannot contain the same entity twice. There must be an attribute that is defined to be unique for every entity instance, so that unique targeting of entities is possible. Some entity types have multiple key attributes. An entity type may also not have a key. This makes it a weak entity.

Value Sets (Domains) of Attributes

Each simple attribute has an associated value set consisting of all values that are possible. Value sets are not displayed in an ER diagram, and they are similar to the basic data types found in most programming languages.

3.4 Relationship Types, Relationship Sets, Roles, and Structural Constraints

Entities can be related to each other. In ER diagrams, we can represent these relationships.

3.4.1 Relationship Types, Sets, and Instances

Just like with entities, a relationship has a certain type that is represented in the ER diagram. Mathematically, the type is a set of relationship that each relate entities.

3.4.2 Relationship Degree, Role Names, and Recursive Relationships

Degree of a Relationship Type

The degree of a relationship type is the number of participating entity types. For most common relationships, this will be two, but it might be more.

Role Names and Recursive Relationships

Each entity type that participates in a relationship type needs to have a role name that defines the role of the entity type in the relationship type. These are normally only necessary in recursive relationship types.

Recursive relationship types are relationships that reference the same entity type on both sides of the relationship type. An example of this is the entity type ENTITY and a relations MANAGER, because a manager is also an employee, and every employee has a manager.

3.4.3 Constraints on Binary Relationship Types

Cardinality Ratios for Binary Relationships

The cardinality ratio of a relationship represents the maximal participation of the relationship. An example is $1:1$, meaning that on both sides of the relationship, only a single entity may participate in the relationship. Other ratios are $1:N$, $N:1$, and $N:M$.

Participation

The participation constraint specifies wether the existance of an entity depends on it being related to another entity. This constraint specifies the minimum number of entities to participate in a relationship. Total participation means that every single entity must participate in the relation. Partial participation means that at least one entity of the entire entity set must participate in the relation; the relationship set can't be empty.

3.4.4 Attributes of Relationship Types

Relationship types can also have attributes, just like entities can. These can for example specify the start date of a relationship.

3.5 Weak Entity Types

Weak entity types do not have a key. Weak entity types are identified by being related to other specific entities that do have keys. This entity type is then called the identifying entity type. A weak entity type always has a total participation constraint to its identifying entity type.

A weak entity type normally does have a partial key, which identifies weak entities that have the same identifying entity (or owner).

3.7 ER Diagrams, Naming Conventions, and Design Issues

3.7.2 Proper Naming of Schema Constructs

Names in an ER-diagram should be chosen to convey as much possible meaning as possible. For entity types, we choose to use singular names. We use the convention that entity types and relations are uppercase, and attributes have their first letter(s) capitalized. Another convention is to choose verbs in relationship types such that the ER-diagram is readable from top to bottom and from left to right.

3.7.3 Design Choices for ER Conceptual Design

Creating a ER-diagram is an iterative process where changes happen until the diagram is complete. Some refinements that are often included are the following:

- An attribute that references another entity can be later included as a binary relationship between the two entity types.
- An attribute that exists in multiple entity types can be promoted into an independent entity type that relates to the other entities.
- A binary relationship type can be demoted into an attribute if it conveys the same meaning and it the cardinality ratio allows this.

3.9 Relationship Types of Degree Higher than Two

3.9.1 Choosing between Binary and Ternary (or Higher-Degree) Relationships

A relationship of degree n will have n edges connecting it to n entities in an ER-diagram. Normally, a relationship type of degree > 2 should only be implemented if it's impossible to implement the same with multiple binary relationships.

3.9.2 Constraints on Ternary (or Higher-Degree) Relationships

A > 2 degree relationship has two constraints. The first is the cardinality, notating a 1, N, or M for every participating entity type. Another is the (min, max) notation, specifiying for each entity the minimum and maximum relationship instances in the relationship set. i

Chapter 4

The Enhanced Entity-Relationship (EER) Model

The ER Modeling concepts discussed in chapter 3 are sufficient for representing many simple database schemas, but are not sufficient for more complex schemas. For this reason, the Enhanced Entity-Relationship Model (EER) was created. This allows for more complex structures in the diagram so mind to diagram mapping can still occur efficiently.

4.1 Subclasses, Superclasses, an Inheritance

The first EER concept we talk about is the subclass and superclass of an entity type. If we have an entity type and we want to subdivide them in different distinct sub-entities, we can use subclasses. An example of a subclass is the entity type EMPLOYEE with subclasses SECRETARY, ENGINEER, and MANAGER. In this case, EMPLOYEE is the so called superclass.

Subclasses inherit from their superclass. This means that any attributes and relationships that belong to the superclass entity type, also belong to the subclass. A subclass together with all attributes an relationships of its superclass can be considered an entity type in its own right.

4.2 Specialization and Generalization

4.2.1 Specialization

Specialization is the process of defining subclasses for an entity type. This set of subclasses is defined on the basis of some distinguishing characteristic of the entities in the superclass.

There are two main reasons for specialization. The first is that certain attributes may only apply to a subclass, and not to the superclass entities. The second is that some relationship types may only be participated in by entities of a certain subclass.

4.2.2 Generalization

We can think of generalization as the reverse process to specialization. In generalization, we take multiple entity types that overlap, and we try to create a superclass that fits them all.

4.3 Constraints and Characteristics of Specialization and Generalization Hierarchies

4.3.1 Constraints on Specialization and Generalization

In some specializations, we can create an attribute on the superclass that defines what subclass this entity belongs to. We call this the defining predicate of the subclass if there is multiple different attributes for this. If there is only one attribute for this, we call it the defining attribute. When there is no condition, the subclass is user-defined.

Two other constraints may apply to a specialization. The first is the disjointness constraint. This constraint specifies that a member of the superclass can only take part in a single subclass. We denote this with a **d** in the circle in the EER diagram. If this constraint does not hold, we denote this with an **o**. The second constraint is the completeness constraint. This works equal to the participation constraint in the ER diagram, but it works on subclass participation instead of relationship participation.

4.3.2 Specialization and Generalization Hierarchies and Lattices

A subclass may itself have subclasses, leading to a hierarchy of subclasses. A shared subclass can also appear, where multiple subclasses specialize into a single one. This creates a lattice structure. The single subclass at the end may then have multiple superclasses, leading to multiple inheritance. This is a feature that isn't supported by all DBMSs, which is something to keep in mind when using multiple inheritance.

4.4 Modeling of UNION Types Using Catagories

Sometimes we need to represent a collection of different entity types as a single entity type. An example is that the owner of a car might be a person, bank, or company; all three of which are distinct entity types. For this we can create a union type, which allows us to create catagories of entities.

In a union type, attribute inheritance works selectively. Only attributes from the current catagory is inherited.

A catagory can be total or partial, where a total catagory holds the union of all the entities in its superclasses, and the partial catagory holds the subset of the union of all the entities.

Part III

The Relational Data Mode and SQL

Chapter 5

The Relational Data Model and Relational Database Constraints

5.1 Relational Model Concepts

The relational data model represent a database as a set of relations. Each relation represents a table of values where each row represents a collection of related data values. A column header represents an attribute, and the table is called a relation.

5.1.1 Domains, Attributes, Tuples, and Relations

A domain D is a set of atomic values. This domain specify basically what can be represented and what cannot. A data type or format is also specified for each domain. From a list of attributes and their domains, we can create a relation schema that can be used to describe a relation. The degree of a relation is equal to the number of attributes in its relation schema.

Each relation has a state that is defined by the n-tuples currently in the relation. If we take the product of the domains of all attributes, we get the set of all possible relation states possible for a relation. If we take the product of the cardinalities of the domains, we get the amount of relation states possible.

5.1.2 Characteristics of Relations

Ordering of Tuples in a Relation

A relation is defined as a set of tuples, that mathematically don't have any order among them. Yet, tuples are stored in some order on disk, and are displayed in some order. This ordering is however not part of the definition.

Values and NULLs in the Tuples

An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown, or that may not apply to a tuple. The exact meaning of NULL can be somewhat hard to know at any point, because NULL has multiple meanings. It has been tried to incorporate different types of NULL, but this has been proven a difficult task.

Interpretation (Meaning) of a Relation

The relation schema can be intepreted as a declaration or assertion. Each tuple in the relation can be seen as a fact, or a particular instance of the assertion.

Some relations may represent facts about entities, while others may represent facts about relationships.

5.2 Relational Model Constraints and Relational Database Schemas

5.2.1 Domain Constraints

Domain constraints specify that any value within a tuple must abide by the domain specified for that attribute.

5.2.2 Key Constraints and Constraints on NULL Values

No two tuples can have the same values for all attributes. Usually, there is a defined subset of attributes that must be unique for any two tuples. We call this a superkey. For every relation, there is at least one superkey, that is, the set of all its attributes. The key of a relation is a minimal superkey, meaning that if any attributes are removed from the key, it is not a superkey anymore.

5.2.3 Relational Databases and Relational Database Schemas

A relatinoal database schema is a set of relation schemas and a set of integrity constraints. When we refer to a relational database, we include both its schema and its current state. A database state that does not obey all the integrity constraints is called not valid. Each DBMS must have a DDL to define a relational database schema.

5.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

The entity integrity constraint states that no primary key value can be NULL. This is because the primary key is used for identification of tuples in a relation.

The referential integrity constraint is specified between two relations and is used to maintain consistency amongs tuples. It states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation.

To describe this more formally, we first need to know what a foreign key is. A foreign key is an attribute in a relation that is also an attribute in another relation. This basically links two tuples together. A foreign key is used to represent relationships in a relational database schema. A foreign key can reference its own relation. For example, an employees manager points to another employee. Both tuples here are employees, which is both the same relation.

5.2.5 Other Types of Constraints

All types of constraints up to now have been so called state constraints. We may also have transaction constraints. These might specify that the value of some attribute in a tuple can only increase.

5.3 Update Operations, Transactions, and Dealing with Constrains Violations

5.3.1 The Insert Operation

The insert operation provides a list of attribute values for a tuple to be inserted into a relation. The insert operation can violate the domain constraint, key constraint, entity integrity constraint, and referential integrity constraint.

5.3.2 The Delete Operation

The delete operation removes a tuple from a relation. This operation can only violate referential integrity. If a delete operation causes a violation, we can handle this by either not allowing the operation, or by cascading. Cascading means to propagate the deletion to referenced tuples as well.

5.3.3 The Update Operation

The update operation updates a value in a relation. This can violate referential integrity and primary key constraints.

Chapter 6

Basic SQL

SQL is the language used for most commercial DBMSs. SQL is a comprehensive database language. It contains statements for data definition, hence, it is a DDL and a DML. SQL also has facilities for defining views, security, and for defining integrity constraints.

6.1 SQL Data Definition and Data Types

6.1.1 Schema and Catalog Concepts in SQL

An SQL schema is identified by a schema name and includes an autorization identifier to indicate the user who owns the schema. Schema elements include tables, types, constraints, views, and domains to describe the schema. A schema is created with the `CREATE SCHEMA` statement as follows:

```
CREATE SCHEMA COMPANY AUTHORIZATION '<user>';
```

6.1.2 The CREATE TABLE Command in SQL

The `CREATE TABLE` command can be used to create a new relation. The attributes are specified with their domains and possible constraints. We also specify which attribute is a key and we specify any foreign keys we might need. Typically, the schema in which tables are defined is implicitly specified by the environment in which the `CREATE TABLE` command is used, but this may be explicitely specifies by writing

```
CREATE TABLE COMPANY.EMPLOYEE
```

as opposed to

```
CREATE TABLE EMPLOYEE
```

6.1.3 Attribute Data Types and Domains in SQL

The following is a list of available data types in SQL. Certain dialects may change this.

- **Numeric** stores a number
- **Character-string** stores a string of characters with n length
- **Bit-string** stores a string of bits
- **Boolean** stores a boolean vaue (true/false)
- **DATE** stores a date in the form `YYYY-MM-DD`
- **timestamp** stores the date and the time

6.2 Specifying Constraints in SQL

6.2.1 Specifying Attribute Constraints and Attribute Defaults

Because NULL values are allowed in SQL, a NOT NULL constraint may be specified on an attribute. This constraint is always implicitly specified for the primary key of a relation. It is also possible to specify a default value for an attribute, which is included in any new tuple where another value is not set for the attribute.

Constraints can also be specified using the `CHECK` clause following an attribute or domain definition. With this we can restrict certain attributes to anything we want. An example that makes sure `Dnumber` falls in [0..20):

```
Dnumber INT NOT NULL CHECK (Dnumber >= 0 AND Dnumber < 20);
```

6.2.2 Specifying Key and Referential Integrity Constraints

We can specify a primary key by saying `PRIMARY KEY` after defining an attribute. The primary key is implicitly defined as unique. We may also define an attribute as unique by saying `UNIQUE` after the definition.

Referential integrity is specified using the `FOREIGN KEY` clause. If an update operation violates the referential integrity, basic SQL behavior is to reject the update. The schema designer can however implement an alternative action to be taken by attatching a referential triggered action clause to any foreign key constraint. These options include `SET NULL`, `CASCADE`, `SET DEFAULT` which must be specified with either `ON UPDATE` or `ON DELETE`.

6.2.3 Giving Names to Constraints

We can give a constraint a name by specifying it directly after the keyword `CONSTRAINT`. A constraint name must be unique for every constraint. A constraint name is used if we later want to change or drop a constraint.

6.2.4 Specifying Constraints on Tuples Using CHECK

Additional table constraints can be specified using `CHECK` at the end of a `CREATE TABLE` statement. These are called row-based constraints, because they apply to each row individually. The `CHECK` clause can also be used to specify more general constraints using the `CREATE ASSERTION` statement in SQL. We discuss this later on.

6.3 Basic Retrieval Queries in SQL

Retrieving information from a database in SQL is done with the `SELECT` statement.

6.3.1 The SELECT-FROM-WHERE Structure of Basic SQL Queries

The basic structure of a `SELECT` statement is the following:

```
SELECT  "attribute list"
FROM    "table list"
WHERE   "condition";
```

Attribute list is a list of attributes that we want the query to return. The table list tells us from what table(s) to retrieve this information, and the condition provides a boolean expression that tells us whether to return a tuple or not. This can be as simple as `TRUE` or `Salary > 60000` but it can also be very complicated, as we will see later.

6.3.2 Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

If we query from multiple tables, we might have attributes with the same name but with different meaning. We can therefore use the table name as a prefix with the attribute. If we have a table `EMPLOYEE` and an attribute named `Name` we could represent it as `EMPLOYEE.Name`.

In the `FROM` cause, we can use the keyword `AS` to import a table under a different name. Expanding the previous example:

```
SELECT  E.Name
FROM    EMPLOYEE AS E
```

We can take this concept further by also specifying different names for the attributes after the `AS` clause:

```
SELECT  E.Fn
FROM    EMPLOYEE AS E(Fn, Ln, Ssn)
```

6.3.3 Unspecified WHERE Clauses and Use of the Asterisk

A `WHERE` clause it not necessarily necessary in SQL. A query without a WHERE clause will just return all tuples that are selected.

To select all attributes from a tuple, we can use an asterisk. Saying `SELECT EMPLOYEE.*` returns all attributes about the employees that meet the requirements given by the `WHERE` clause.

6.3.4 Tables as Sets in SQL

In SQL, a table is not represented as a set but as a multiset, meaning duplicate tuples can appear. To remove duplicate queries from a query result, we can use the `DISTINCT` keyword before specifying what to select. This removes any duplicate tuples from the result.

Other multiset operations are `UNION ALL`, `EXCEPT ALL`, and `INTERSECT ALL`.

6.3.5 Substring Pattern Matching and Arithmetic Operators

We can use the keyword `LIKE` to do pattern matching. Here a `%` replaces an arbitrary number of zero or more characters, and the _ replaces a single character.

6.3.6 Ordering of Query Results

A query result can be ordered by using the `ORDER BY` clause at the end of the query. We can use the keywords `ASC` and `DESC` to change if it uses ascending or descending order.

6.4 INSERT, DELETE, and UPDATE Statements in SQL

6.4.1 The INSERT Command

The `INSERT` command is used to add a single tuple to the a relation. We can either manually set values to insert, or we might use the result of another query. An example of this can be seen on page 229 of the book.

6.4.2 The DELETE Command

The `DELETE` command is used to delete tuples from a relation. We must specify where to delete something from and what the condition is for deletion.

6.4.3 The UPDATE Command

The `UPDATE` command is used to change a tuple in a relation. We specify where to update something, what to update and to what it should be updated, and what the condition is for updating.

Chapter 7

More SQL: Complex Queries, Triggers, Views, and Schema Modification

7.1 More Complex SQL Retrieval Queries

7.1.1 Comparisons Involving NULL and Three-Valued Logic

When a NULL value is involved in a comparison operation, the result is said to be unknown. We can have comparisons with true, false, and NULL, hence SQL uses a three-value logic.

In queries, the general rule is that only tuples where the `WHERE` clause evaluates to true are selected.

7.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require existing values from the database to be used in a comparison. For this we can use nested queries, where the `WHERE` clause contains another query. We use the keyword `IN` for comparison. If we are certain that the query returns a single value, then it is also permissable to use = instead.

7.1.3 Correlated Nested Queries

Whever a condition in the `WHERE` clause of a nested query references an attribute from the outermost query, this is said to be a correlated nested query. In general, a query with a nested select-from-where block using `IN` or = can always be rewritten as a single block query.

7.1.4 The EXISTS and UNIQUE Functions in SQL

The `EXISTS` function in SQL tells us wether a select query has returned anything or not. This can be used in a `WHERE` clause in our outermost query.

Another function `UNIQUE` returns true if there is no duplicate tuples and false if there is. This can be used to test of a query result is a set or a multiset.

7.1.5 Explicit Sets and Renaming in SQL

It is possible to use explicit values in a `WHERE` clause instead of a nested query. An example:

```
SELECT  DISTINCT Essn
FROM    WORKS_ON
WHERE   Pno IN (1,2,3);
```

We can also us the `AS` keyword to rename any attribute that we select. The name will then be reflected differently in the output table.

7.1.6 Joined Tables in SQL and Outer Joins

We can join tables in the `FROM` clause of a query. This allows us to join two tables together on a specific attribute and then return this to the user. There are a couple types of join:

- `NATURAL JOIN` no join condition is specified; the join happens on attributes with the same name
- `INNER JOIN` the default join type. A condition is specified; a tuple is included in the result iff a matching tuple exists in the other relation
- `OUTER JOIN` can be left, right, or full. Includes all tuples from the chosen side, no matter if they can be joined or not

7.1.7 Aggregate Functions in SQL

Aggregrate functions are used to summarize information. The functions are `SUM`, `MAX`, `MIN`, and `AVG`.

7.1.8 Grouping: The GROUP BY and HAVING Clauses

If we want to apply aggregrate functions on subgroups of tuples returned, we can use the `GROUP BY` keyword. If we have a query where we select a department number and the average salary, saying `GROUP BY Dno` gives us the average salary per department.

Together with `GROUP BY`, we can also use `HAVING`. This is a condition that's applied on the `GROUP BY`, where groups that don't satisfy the `HAVING` clause are discarded.

7.1.9 Other SQL Constructs: WITH and CASE

The `WITH` clause is used to specify a table that is used only for this query. It is somewhat similar to defining a view. SQL also has a `CASE` statement, where we can choose to do something in some cases. For examples on both these constructs, see page 252 in the book.

7.2 Specifying Constraints as Assertions and Actions as Triggers

7.2.1 Specifying General Constraints as Assertions in SQL

In SQL we can create our own constraints. A way to do this is by specifying an assertion, using `CREATE ASSERTION`. An assertion specifies a query that returns true or false, and it must return true for the database state to be valid.

7.2.2 Introduction to Triggers in SQL

In SQL we can also create triggers using `CREATE TRIGGER`. A trigger specifies the action to be taken when a certain event occurs.

A trigger might be `BEFORE`, `ON`, or `AFTER` some action and then specifies a condition and action to be taken when the condition is met.

7.3 Views (Virtual Tables) in SQL

7.3.1 Concept of a View in SQL

A view in SQL is a table that is derived from other tables. It is a specification of a table that may not exist pysically, but used data obtained from other tables.

7.3.2 Specification of Views in SQL

We specify a view with the `CREATE VIEW` command. We give the view a name, and we tell it what attributes it needs by specifying it to be the result of a select-from-where query. If we modify the tables where the view is based from, the view needs to automatically update. This is the responsibilty of the DMBS.

To remove a view, we can use the `DROP VIEW` command.

7.3.4 Views as Authorization Mechanisms

We can create views that can only see a subset of data that we have. This allows us to give a user permission to see a view, but not the base table. We can use this for autorization, so a user can only see precisely what they are allowed to.

7.4 Schema Change Statements in SQL

7.4.1 The DROP Command

The `DROP` command is used to drop named schema elements. We can drop a schema, or a table, or anything else that's named. We may also specify `RESTRICT` or `CASCADE`. These meaning to only drop if the schema is empty or table isn't referenced in any constraints, or to drop every constraint, view, and other referencing element as well, respectively.

7.4.2 The ALTER Command

We can alter a table or schema using the `ALTER` command. We may for example add a column to a table, drop a column, or alter a column to change a constraint or the default value.

Chapter 9

Relational Database Design by ER- and EER-to-Relational Mapping

In this chapter there were originally a lot of figures to help with explaining, these are not included for the summary. Anyone reading this that doesn't understand it without examples is prompted to go and read the book

9.1 Relational Database Design Using ER-to-Relational Mapping

9.1.1 ER-to-Relational Mapping Algorithm

Step 1: Mapping of Regular Entity Types

For each regular entity E in the ER schema, create a relation R that contains all the simple attributes of this entity. Choose one of the key attributes of E as the primary key of R.

Step 2: Mapping of Weak Entity Types

For each weak entity type W with owner type E, create a relation R and include all the simple attributes of W in R. The primary key of R correspond to E.

Step 3: Mapping of Binary 1:1 Relationship Types

Can be done in 3 different ways:

- **Foreign key approach** Choose one of the relations(S) and include as a foreign key in S the primary key of the other relation (T). If there is an entity type with total participation, this should take the role of S. Include all simple attributes of the relationship as attributes of S.
- **Merged relation approach** When both participations are total, we can merge the two relations into one.
- **Cross-reference or relationship relation approach** We can create a third relation R for the purpose of cross-referincing the primary keys of relations S and T.

Step 4: Mapping of 1:N Relationship Types

Can be done in 2 ways:

- **Foreign key approach** This is the same as the foreign key approach in step 3, except that the 1-side is now R and the N-side is S.
- **Relationship relation approach** We create a seperate relation R whose attributes are the primary keys of S and T. The primary key of R is the primary key of S.

Step 5: Mapping of Binary M:N Relationship Types

The only option for M:N relationships is to use the Relationship relation approach.

Step 6: Mapping of Multivalued Attributes

For each multivalued attribute A, create a relation R. R will include an attribute corresponding to A, plus an attribute that is a foreign key of the relation that has A as an attribute.

Step 7: Mapping of N-ary Relationship Types

For this we also use the Relationship relation approach, just with more than 2 attributes.

9.2 Mapping EER Model Constructs to Relations

9.2.1 Mapping of Specialization or Generalization

We have 4 options to represent subclasses:

1. Create a relation for the superclass and one for each of the subclasses. Use a foreign key to link them together.
2. Create a seperate relation per subclass with each all the attributes of their respective subclass and the superclass.
3. Take all attributes from the subclasses and put them together in one relation with the superclass. Non-applicable subclasses' attributes will be NULL.
4. The same as option 3 but include boolean flags to set which specialization is true for which tuple.

9.2.2 Mapping of Shared Subclasses (Multiple Inheritance)

Options 3 and 4 of section 9.2.1 are applicable to use here.

9.2.3 Mapping of Categories (Union Types)

Step 9: Mapping of Union Types (Catagories)

We can define this using a surrogate key. This is a relation that consists of only a key. This key is a foreign key for, on one side, the relation, and on the other side all the catagories that are each a seperate relation.