CSE1500 Database Technology Midterm

Contents

Chapter1. Databases and Database Users

An app where operations with a database are central it's called a database application. New platforms have pushed the creation of databses that store nontraditional data (images, video, etc. instead of plain text and numbers).

New types of database systems are often referred to as big data storage systems or NOSQL systems, which have been created to manage data for social media applications, and are used by companies such as Google, Amazon and Yahoo to manage the data requried in their web search engines as well as to provide cloud storage.

OLAP = online analytical processing systems used to extract and analyze useful business information from large databases to support decision making.

real-time active and active database technology = used to control industrial and manufacturing processes

database search techniques = used by search engines.

database: collection of related data. Furthermore, a database has the following implicit properties:

1. It **represents an aspect of the real world**, and changes of the world are reflected in the database.
2. It is a **logically coherent collection of data** with some inherent meaning. Random assortment of data cannot be referred to as a database
3. It is designed, built and populated with data **for a specific purpose**. It has an intended group of **users** and an application in which the users are interested.

A database can be of any size and complexity.

A database can be distributed over several servers and be acessed by millions of users.

A database can be manually mantained (such as a library card system) or computeried (the same thing but digital).

A **database management system** (DBSM) is a computerized system that enables users to create and mantain a database. It is a **general purpose software system** that facilitates the process of

defining: specifying data types, structures and constrains. This descriptive information is stored by the DMS in the form of a "database catalog/dictionary" called meta-data (which can be accessed by queries to skip things and be faster).

constructing: storing the data on some storage medium

manipulating: querying (SCRUD), and reports.

and sharing databses among users and applications.

a transaction is a type of query that makes a simultaneous read and write in different locations.

the DBMS also **protects** (crashes and security) and **mantains** (allow for updates and changes) the database.

DBMS is often shortened as **database system**.

Design of a new application for an existing database or design of a brand new database starts off with a phase called **requirements specification and analysis.** These requirements are documented in detail and transformed into a conceptual design that can be represented and manipulated using some computerized tools so that it can be easily maintained, modified, and transformed into a **database implementation.**

The design is then translated to a **logical design** that can be expressed in a data model implemented in a commercial DBMS. The final stage is **physical design**, during which further specifications are provided for storing and accessing the database

Characteristics of the database approach

traditional file processing: each user defines and implements files needed for a specific software application as part of programming the application (i.e. container classes in java). Different users may required different data, that might be related in some sort of way. But you don't want to constantly merge data into new objects ad-hoc per query.

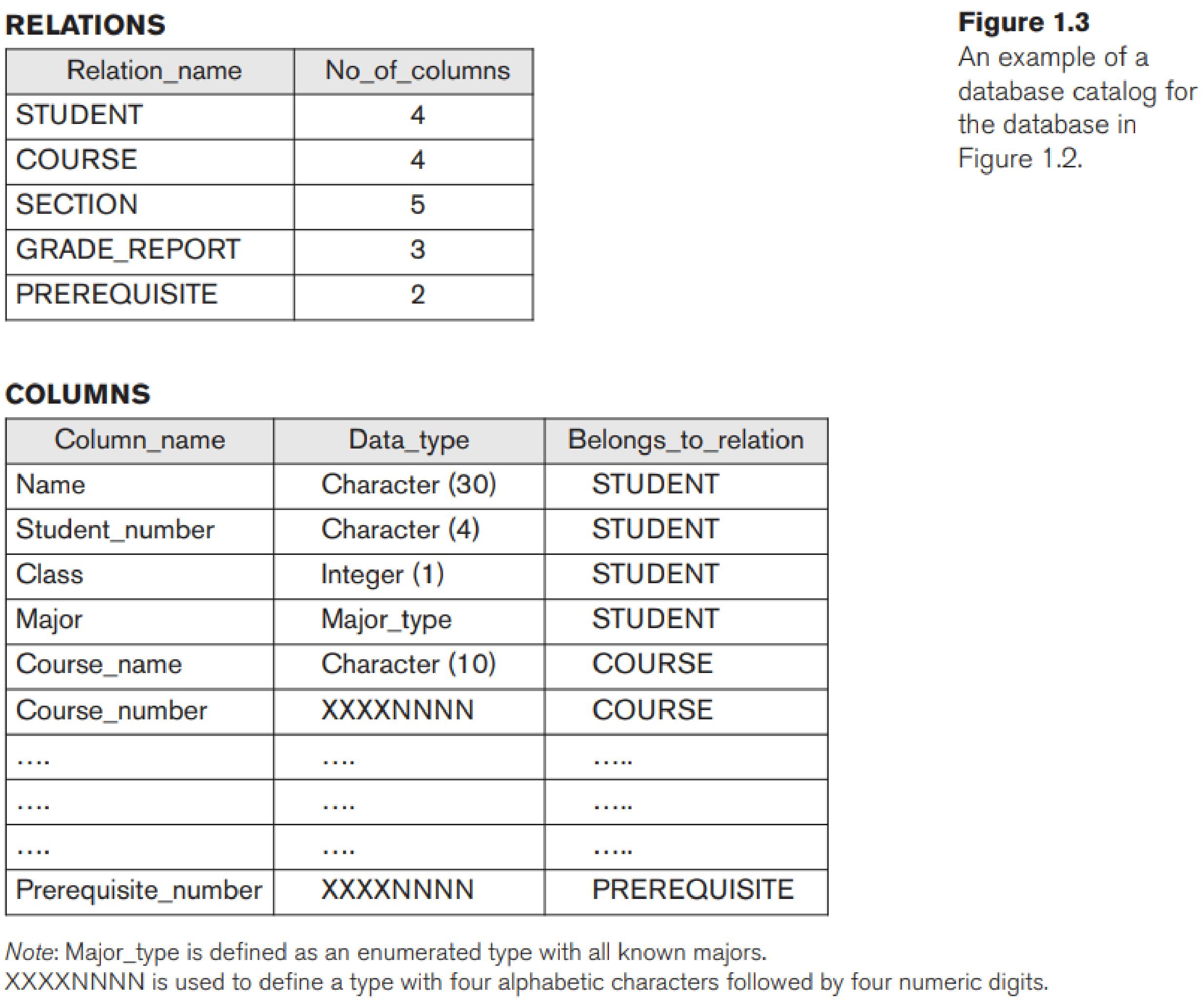
Instead, the **database approach** has a single repository that mantains the data, where it has been defined once and then such data may be accesed by various users, repteadly, through queries, transactions and application programs. A database approach has:

1. self-describing nature of the database system
2. Insulation between programs and data
3. Support of multiple views of the data
4. Sharing of data and multiuser transaction processing

Self-Describing Nature of a Database System (Meta-Data)

Database systems not only contain the database itself, but also a complete definition or description of the database structure and constraints, called "meta-data" which is stored in the DBMS catalog, and contains the structure of each file, type and storage format, and data constraints. Newer systems such as NOSQL do not require meta-data because it is "self describing".

DBMS software can access diverse databases by extracting the database definitions from the catalog and using these definitions.



RELATIONS

Relation_name	No_of_columns
STUDENT	4
COURSE	4
SECTION	5
GRADE_REPORT	3
PREREQUISITE	2

COLUMNS

Column_name	Data_type	Belongs_to_relation
Name	Character (30)	STUDENT
Student_number	Character (4)	STUDENT
Class	Integer (1)	STUDENT
Major	Major_type	STUDENT
Course_name	Character (10)	COURSE
Course_number	XXXXNNNN	COURSE
….	….	…..
….	….	…..
….	….	…..
Prerequisite_number	XXXXNNNN	PREREQUISITE

Note: Major_type is defined as an enumerated type with all known majors.
XXXXNNNN is used to define a type with four alphabetic characters followed by four numeric digits.

Figure 1.3
An example of a database catalog for the database in Figure 1.2.

Data Abstraction

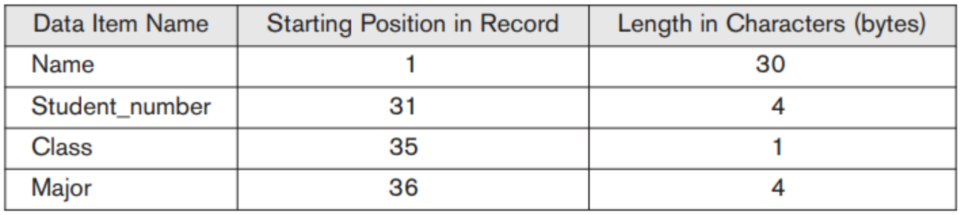
Program-data independence

We only need to change the description of a table in the catalog instead of rwriting an entire program:

In traditional file processing, the structure of data files is embedded in the application programs, so any changes to the structure of a file may require changing all programs that access that file. By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs.

Program-operation independence

An operation (also called a function or method) is specified in two parts. The interface (or signature) of an operation includes the operation name and the data types of its arguments (or parameters). **The implementation (or method) of the operation is specified separately and can be changed without affecting the interface**. **User application programs can operate on the data by invoking these operations through their names and arguments**, regardless of how the operations are implemented.



Data Item Name	Starting Position in Record	Length in Characters (bytes)
Name	1	30
Student_number	31	4
Class	35	1
Major	36	4

Figure 1.4
Internal storage format for a STUDENT record, based on the database catalog in Figure 1.3.

Data model

It is a very used term with different definitions, but in the context of data abstraction a data model is a simplified visual representation of a data abstraction by using logic symbols and shapes to define relationships and properties of objects (such as ER or UML models).

SQL is the standard data model and language for relational databases.

NOSQL is generally interpreted as Not Only SQL.

Views

A DBMS has support for views, a view is a read-only representation of database contents that is originated from a set of SELECT statements often combined with JOIN.

Multiuser Transaction Processing

Online transaction processing (OLTP) applications: A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include concurrency control software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct. For instance a DBMS should ensure that each flight seat can be accessed by only one agent at a time for assignment to a passenger.

Transaction: A transaction is an executing program or process that includes one or more database accesses, such as reading or updating of database records. It has the following properties:

1. **Isolation:** The isolation property ensures that each transaction appears to execute in isolation from other transactions, even though hundreds of transactions may be executing concurrently
2. **Atomicity:** either all the database operations in a transaction are executed or none are. (i.e. transfering money from one account to another can’t be allowed to be half finished, either money leaves account A AND arrives account B or no money leaves A AND no money arrives B)

Database stakeholders

Database Administrators (DBA)

Administrates the database environment: the database itself (primary resource) and the DBMS and related software (secondary resource).

The admin authorizes access to the database, coordinates and monitors its use, mantains the database with software and hardware updates, responsible for security and response time.

Database Designer

Choose appropiate structures to represent and store the data, before the database is actually implemented and populated. It coordinates the requirements and makes a design that meets them.

End Users

They access the database for querying, updating, and generating reports.

- **Casual end users**: Use a sophisticated database query interface to specify their requests and a typically middle or high level managers
- **Naive/parametric end users**: They make queries and update the database using standard types of queries and updates (called canned transactions) that have been already programmed and tested (so they use a GUI, they dont write the query themselves)
- **Sophisticated end users**: engineers, scientists, business analysts who directly use the DBMS for their own uses.
- **Standalone users**: mantain personal databases by using ready-to-use packages with GUIs.

System analysts

Determien the requirement of end users and develop specifications for standard canned transactions to meet these requirements.

Application programmers/Software developers/Software engineers

Implement these specifications as sprograms (test, debug, comment and mantain too) and are fully familiar with the capabilities by the DBMS.

DBMS system designers and implementers

Design and implement the DBMS modules and interfaces as a software package.

Tool developers

design and implement tools—the software packages that facilitate database modeling and design, database system design, and improved performance.

Operators and mantianence personal

(system administration personnel) are responsible for the actual running and maintenance of the hardware and software environment for the database system.

Chapter 2. Database System concepts and Architecture

Modern DBMS packages are modular in design, with a client/server system architecture. Old ones where a tightly integrated system.

The rapid growth of data has promoted the trends in computing where large centralized mainframe computers are being replaced by hundreads of distributed workstations.

Cloud computing environments consist of thousand of large servers managing big data for users on the Web.

In a basic client/server DBMS architecture, the system functionality is distributed between two types of modules. A **client module** is typically designed so that it will run on a mobile device, user workstation, or personal computer (PC). Usuall include user-friendly interfaces such as apps for mobile devices, or forms- or menubased GUIs (graphical user interfaces) for PCs. A **server module** typically handles data storage, access, search, and other functions.

2.1 Data Models, Schemas and Instances

A data abstraction is a **highlight of the essential features** of the data of a database. A data model is a mean to represent a data abstraction.

Data models show the structure of a database, which constitutes, among other things, the data types, relationships, and constraints that apply to the data.

2.1.1 Categories of Data Models

- **High level/Conceptural data models:** define high level concepts perceived mostl at the front-end experience of its users. Ubckyde cibceots such as **entities** (represent a real world-object or concept i.e. 'employee', 'supplier'), **atributes** (properties that describe an entity i.e. 'name', 'salary'), **relationships** (i.e. 'project' and 'consultant').
- **Low level/Physical data models:** describe the details of how data is stored (on the disk) as files in the computer by representing information such as record formats, record orderings, and access paths, these details are not meant for the end users. **acces path:** is **a search structure that makes the search for particular database records efficient, such as indexing or hashing.**
- **Representational/implemention data models:** mid-level description of how data is organized. common in traditional commercial DBMS, include the relational data model, as well as legacy data models (network and hirearchical models commonly used in the past). Representational data models represent data by using record structures and sometimes are called record-based data models.
- **Object data model:** new higher-level implmenation model closer to conceptual models, often referred to as ODMG (Object Data Model Group)
- **Self-describing data models:** The data storage in systems based on these models combines the description of the data with the data values themselves. In traditional DBMSs, the description (schema) is separated from the data. These models include **XML**, as well as many of the key-value stores and NOSQL systems that were recently created for managing big data.

2.1.1 Schemas, Instances, and Database State

Schema: description of a database. It is specified during database design adn it is not expected to change very often. A schema diagram displays the structure of each record type but not the actual instances of records. A schema diagram displays only some aspects of a schema, such as the names of record types and data items, and some types of constraints

Each object in the schema (such as STUDENT or COURSE) is a schema construct.

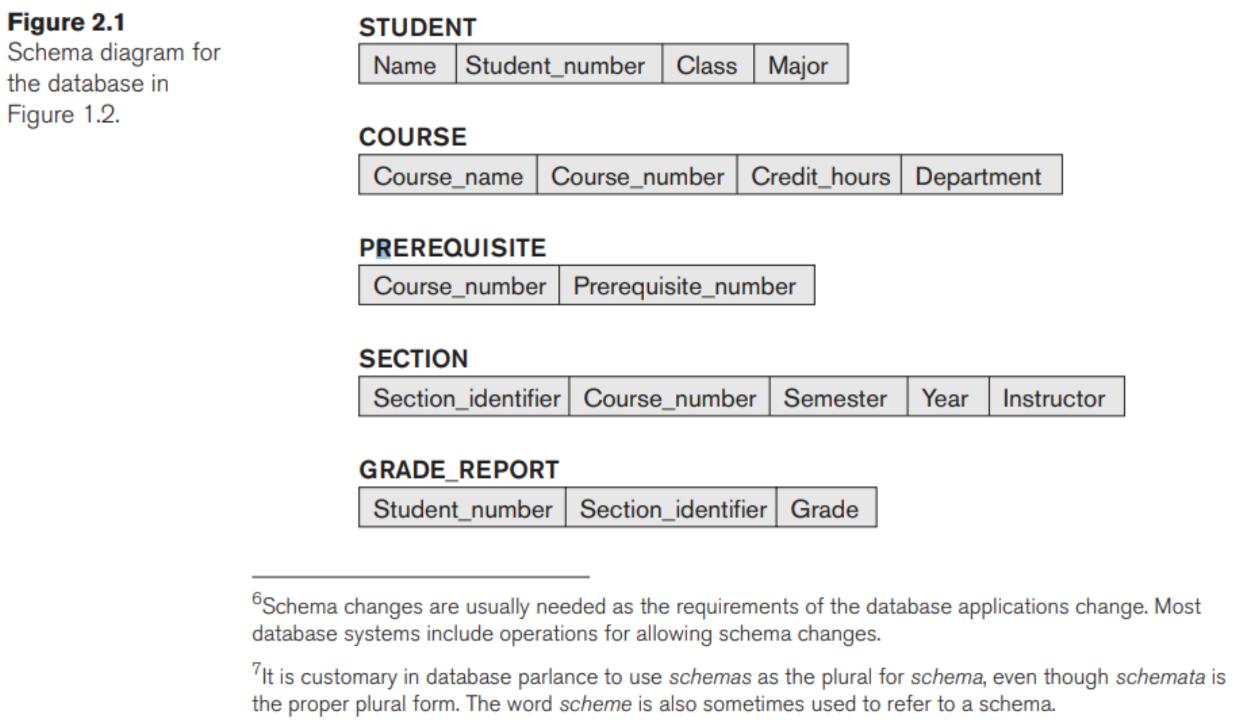


Figure 2.1
Schema diagram for the database in Figure 1.2.

STUDENT

Name	Student_number	Class	Major

COURSE

Course_name	Course_number	Credit_hours	Department

PREREQUISITE

Course_number	Prerequisite_number

SECTION

Section_identifier	Course_number	Semester	Year	Instructor

GRADE_REPORT

Student_number	Section_identifier	Grade

[6] Schema changes are usually needed as the requirements of the database applications change. Most database systems include operations for allowing schema changes.

[7] It is customary in database parlance to use *schemas* as the plural for *schema*, even though *schemata* is the proper plural form. The word *scheme* is also sometimes used to refer to a schema.

The data in the database at a particular moment in time is called a database state or snapshot. It is also called the current set of occurrences or instances in the database. Every time we insert or delete a record or change the value of a data item in a record, we change one state of the database into another state.

When we define a new database, we specify its database schema only to the DBMS. the corresponding database state is the empty state with no data.

We get the initial state of the database when the database is first populated or loaded with the initial data. From then on each new modification of the data represents a new database state.

The DBMS is partly responsible for ensuring that every state of the database is a valid state—that is, a state that satisfies the structure and constraints specified in the schema.

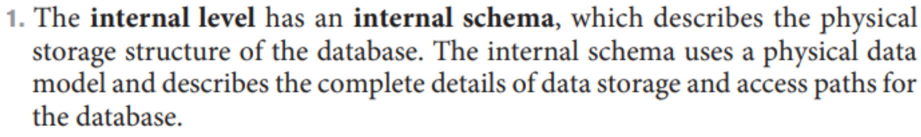
The schema is sometimes called the intension, and a database state is called an extension of the schema.

Making changes in let’s say STUDENT and adding another field suchas Email is known as schema evolution.

2.2 Three-Schema Architecture and Data Independence

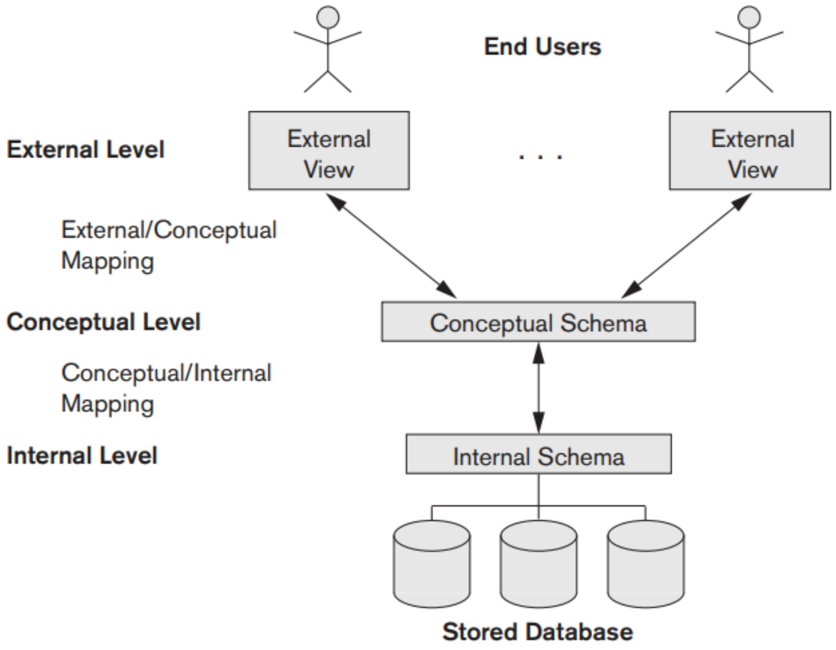
1. **Use a catalog to store the database description (Schema) so as to make it self-desribing**
   The three schemas are only descriptions of data; the actual data is stored at the physical level only. The processes of transforming requests and results between levels are called mappings.
2. **Insulation of programs and data**
3. **Support of multiple user views**

Architecture, low/physical level:



1. The **internal level** has an **internal schema**, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

Figure 2.2
The three-schema architecture.



[9] This is also known as the ANSI/SPARC (American National Standards Institute/ Standards Planning And Requirements Committee) architecture, after the committee that proposed it (Tsichritzis & Klug, 1978).

Conceptual Level

The conceptual level has a conceptual schema, which describes the structure of the whole database for a community of users. This implementation is often based on a conceptual schema design in a high-level data model.

View Level

The external or view level includes a number of external schemas or user views. , each external schema is typically implemented using a representational data model, possibly based on an external schema design in a high-level conceptual data model

2.2.2 Data Independence

The capacity to change the schema at one level of a database system without having to change the schema at the next higher level.

1. Logical data independence is the capacity to change the conceptual schema without having to change external schemas or application programs.
2. Physical data independence is the capacity to change the internal schema without having to change the conceptual schema.

2.3 Database Languages and Interfaces

2.3.1 DBMS Langauges

One language, called the **data definition language** (**DDL**), is used by the DBA and by database designers to define conceptual and internal schemas. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.

The DDL is used to specify the conceptual schema only. Another language, the **storage definition language** (**SDL**), is used to specify the internal schema.

In most relational DBMSs today, there is no specific language that performs the role of SDL. Instead, the internal schema is specified by a combination of functions, parameters, and specifications related to storage of files.

View definition language (**VDL**): to specify user views and their mappings to the conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas.

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database, The most common “SCRUD” select, create, read, update, delete are supported by the **data manipulation language** (**DML**). **SQL** is the most common relational database language, which represents a **combination** of **DDL, VDL and DML**. The SDL was a component in early versions of SQL but has been removed from the language to keep it at the conceptual and external levels only.

A query in a high-level DML often specifies which data to retrieve rather than how to retrieve it; therefore, such languages are also called declarative.

A high-level DML used in a standalone interactive manner is called a query language.

2.3.2 DBMS Interfaces

User-friendly interfaces provided by a DBMS may include the following:

1. Menu-based Interfaces for Web Clients or Browsing.
2. Apps for Mobile Devices
3. Forms-based Interfaces
4. Graphical User Interfaces
5. Natural Language Interfaces
6. Keyword-based Database Search
7. Speech Input and Output
8. Interfaces for Parametric Users (small commands to minimize data entry time)
9. Interfaces for the DBA (that support his exclusive admin rights activites)

2.4 Database System Environment

2.4.1 DBMS Component Modules

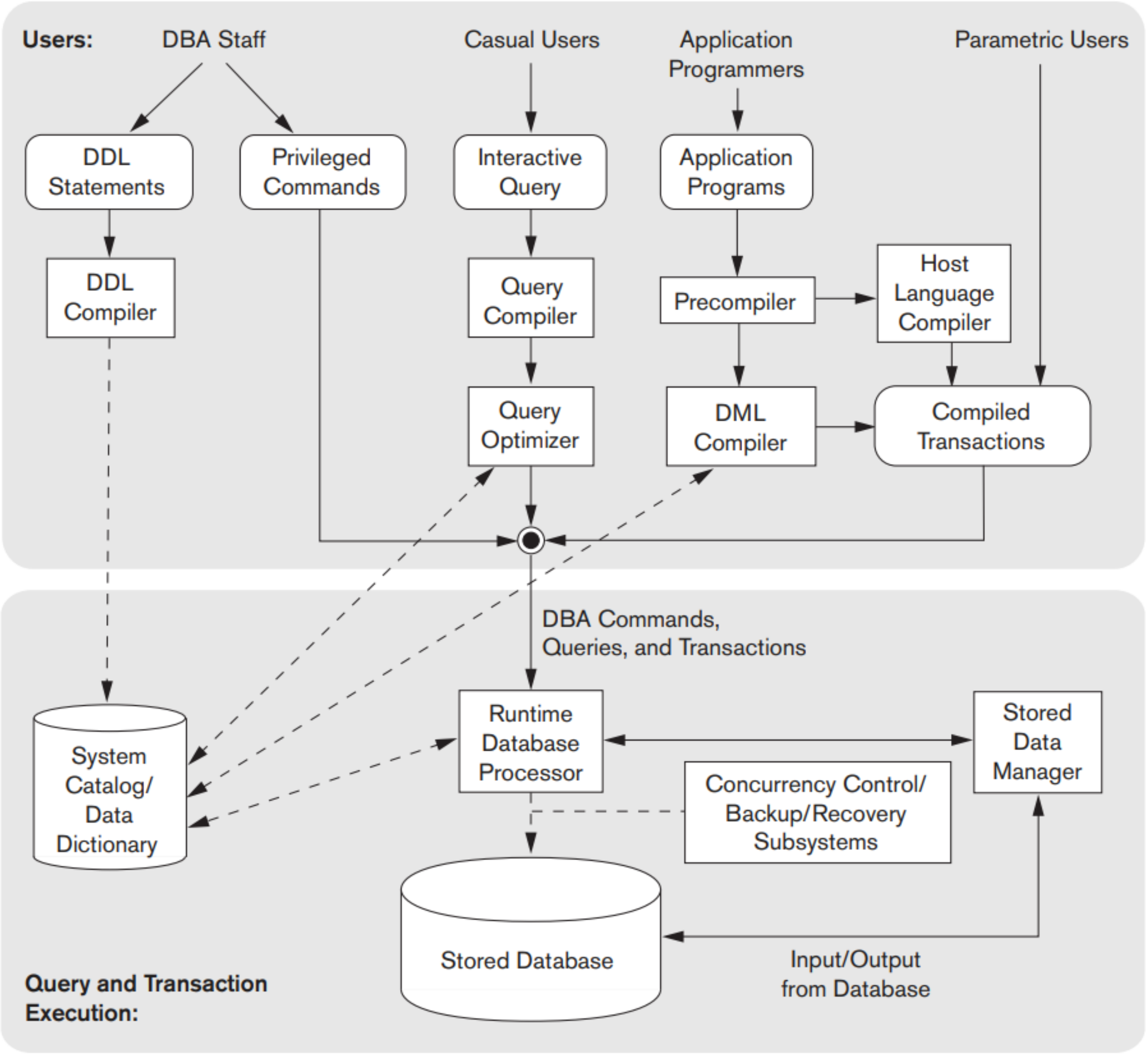


Figure 2.3
Component modules of a DBMS and their interactions.

2.4.2 Database System Utilities

1. **Loading:** A loading utility is used to load existing data files—such as text files or sequential files—into the database.
2. **Backup:** A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape or other mass storage medium.
3. **Database storage reorganization:** This utility can be used to reorganize a set of database files into different file organizations and create new access paths to improve performance.

4. **Performance monitoring:** monitors database usage and provides statistics to the DBA.

2.5 Centralized and Client/Server Architectures for DBMSs

Centralized architecture

At the beginning the DBMS itself was centralized where the DBMS functionality, application program execution, and user interface processing were carried out on one machine. Allocating all of the processing power at the DBMS side (people did not have resourceful PCs back then and connected to the DBMS via terminals so mostly Display operations were carried at the user side).

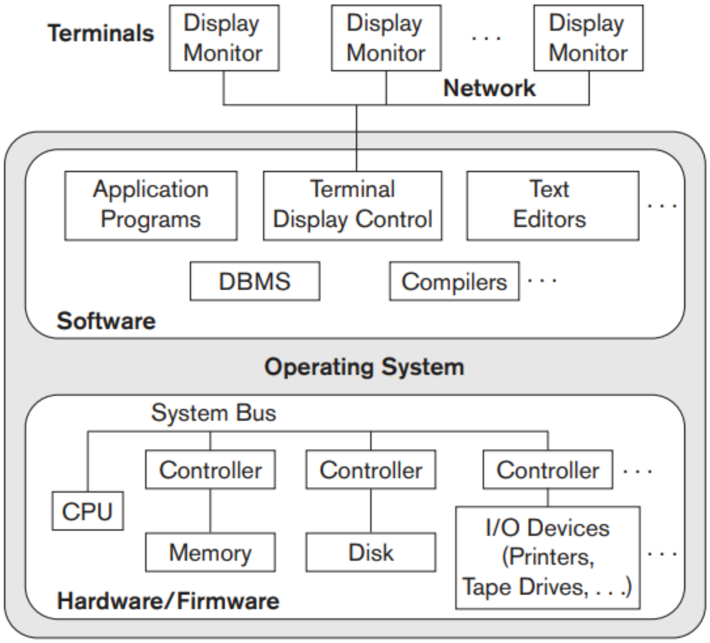


Figure 2.4
A physical centralized architecture.

Client/Server architecture

The client/server architecture was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers, Web servers, e-mail servers, and other software and equipment are connected via a network.

The idea is to define specialized servers with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a file server that maintains the files of the client machines. Another machine can be designated as a printer server by being connected to various printers; all print requests by the clients are forwarded to this machine. Web servers or e-mail servers also fall into the specialized server category. The resources provided by specialized servers can be accessed by many client machines. The client machines provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications.

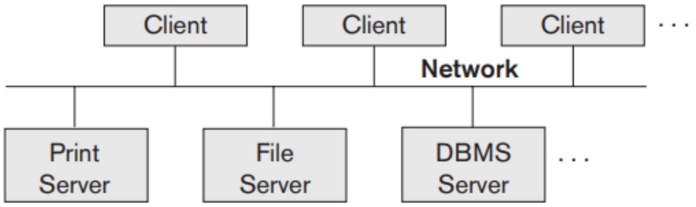


Figure 2.5
Logical two-tier client/server architecture.

Two-tier architectures (use APIs)

A standard called Open Database Connectivity (ODBC) provides an application programming interface (API), which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed.

A related standard for the Java programming language, called JDBC, has also been defined. This allows Java client programs to access one or more DBMSs through a standard interface.

Three-tier acrchitectures

Adds an intermediate layer between the client and the database server.

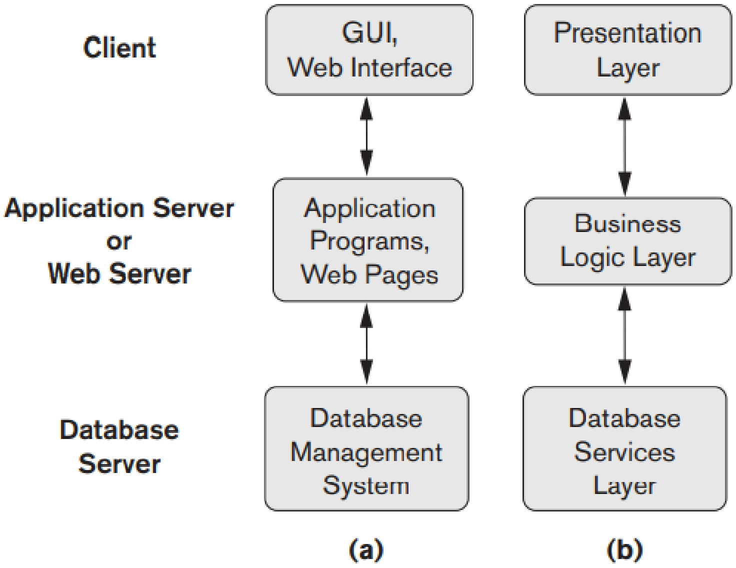


Figure 2.7
Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.

2.6 Categories of DBMS

We can categorize DBMSs based on the **data model**: relational, object, object-relational, NOSQL, key-value, hierarchical, network, and other (such as tree-structured XML data model).

The second criterion used to classify DBMSs is the **number of users** supported by the system. Single-user systems support only one user at a time and are mostly used with PCs. Multiuser systems, which include the majority of DBMSs, support concurrent multiple users.

The third criterion is the number of sites over which the database is **distributed**. A DBMS is centralized if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database reside totally at a single computer site. A distributed DBMS (DDBMS) can have the actual database and DBMS software distributed over many sites connected by a computer network. Big data systems are often massively distributed, with hundreds of sites. The data is often replicated on multiple sites so that failure of a site will not make some data unavailable.

The fourth criterion is **cost**. It is difficult to propose a classification of DBMSs based on cost. Today we have open source (free) DBMS products like MySQL and PostgreSQL that are supported by third-party vendors with additional services.

We can also classify a DBMS on the basis of the **types of access path options for storing files**. One well-known family of DBMSs is based on inverted file structures.

Finally, a DBMS can be **general purpose or special purpose**. When performance is a primary consideration, a special-purpose DBMS can be designed and built for a specific application; such a system cannot be used for other applications without major changes. Many airline reservations and telephone directory systems developed in the past are special-purpose DBMSs. These fall into the category of online transaction processing (OLTP) systems, which must support a large number of concurrent transactions without imposing excessive delays.

Chapter 5. The Relational Data Model and Relational Database Constraints

5.1 Relational Model Concepts

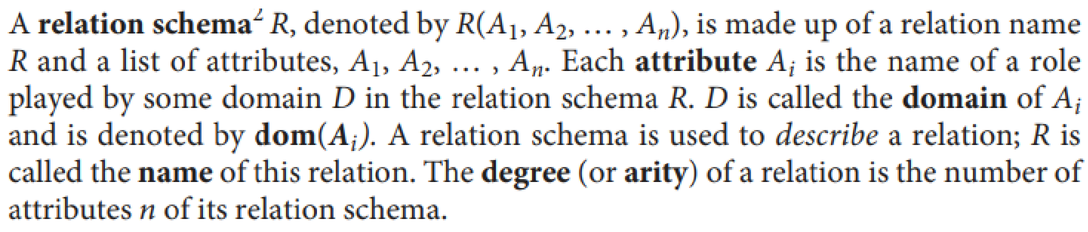
The relational model represents the database as a collection of relations.

Each relation resembles a table of values or, to some extent, a flat file of records.

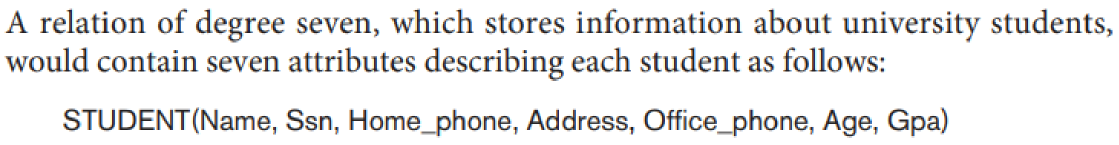
When a relation is thought of as a table of values, each row in the table represents a record, which multiple attributes (columns).

5.1.1 Domains, Attributes, Tuples, and Relations

A domain D is a set of atomic values. By atomic we mean that each value in the domain is indivisible as far as the formal relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. For example, the column with header Usa_phone_number, has a domain of the set of ten-digit phone numbers valid in the United States. This is a logical defintion of a domain. A data type or format is also specified for each domain. For example, the data type for the domain Usa_phone_numbers can be declared as a character string of the form (ddd)ddd-dddd, where each d is a numeric (decimal) digit and the first three digits form a valid telephone area code. **A domain is thus given a name, data type, and format.**



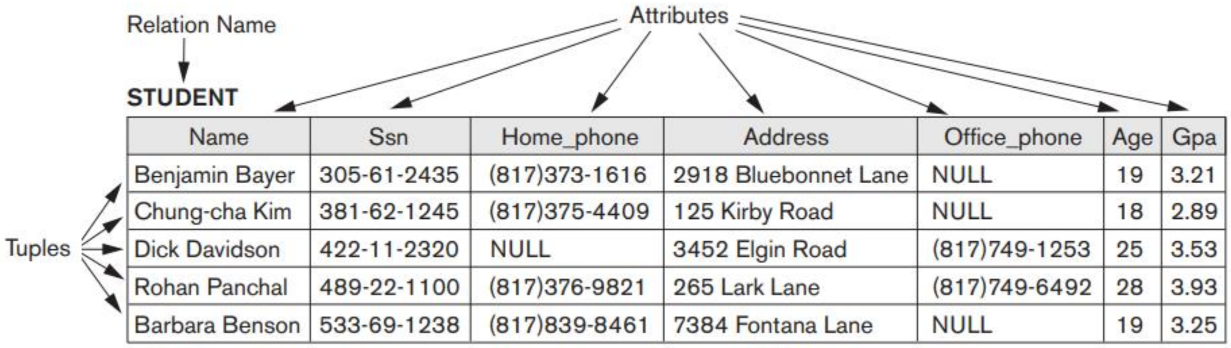
A **relation schema**[2] R, denoted by $R(A_1, A_2, \ldots, A_n)$, is made up of a relation name R and a list of attributes, $A_1, A_2, \ldots, A_n$. Each **attribute** A_i is the name of a role played by some domain D in the relation schema R. D is called the **domain** of A_i and is denoted by **dom**(A_i). A relation schema is used to *describe* a relation; R is called the **name** of this relation. The **degree** (or **arity**) of a relation is the number of attributes n of its relation schema.



A relation of degree seven, which stores information about university students, would contain seven attributes describing each student as follows:

STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)

So this relations mean that all the attributes belong to (and define) the same “relation schema” student, aka student entity.


Relation Name
Attributes
STUDENT
Tuples

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25

Figure 5.1
The attributes and tuples of a relation STUDENT.

Tuples = records

5.1.2 Characteristics of Relations

Tuples in a relation do not have any particular order. That is, records in a table for student entity don't have a particular order. In a file records are physicall stored on disk so the have an intrinsic order.

The definition of a relation does not specify any order. When a relation is implemented as a file or displayed as a table, a particular ordering may be specified on the records of the file or the rows of the table.

A mapping from R to D is the union of these 2 sets. a tuple can be considered as a set of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute Ai to a value vi from dom(Ai) The ordering of attributes is not important, because the attribute name appears with its value

The first normal form assumption, the so called "flat relational model" implies that all records are composed of the exact same attributes, if for any reason a student doesn't have an office phone, the record still has the Office_phone attribute and the NULL value is used to denote what 'null' means in any other programming language.

5.1.3 Relational Model Notation

- A relation schema (for an entity, such as student) R of degree n is denoted by R(A_1, A_2, … , A_n) (A is the attribute/field/column).
- The uppercase letters Q, R, S denote relation names
- The lowercase letters q, r, s denote relation states.
- The letters t, u, v denote tuples.

5.2 Relational Model Constraints and Relational Database Schemas

Constraints on databases can generally be divided into three main categories:

1. Constraints that are **inherent in the data model**. We call these inherent model-based constraints or **implicit constraints**.
2. Constraints that can be **directly expressed in the schemas of the data model**, typically by specifying them **in the DDL** (data definition language, see Section 2.3.1). We call these **schema-based constraints** or **explicit constraints**.

3. Constraints that **cannot be directly expressed in the schemas of the data mod**el, and hence must be expressed and **enforced by the application programs or in some other way**. We call these application-based or semantic constraints or business rules.

5.2.1 5.2.2 Key Constraints and Constraints on NULL Values

See 5.1.1 for normal attributes constraints (mostly data type, size, and format).

By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. No two tuples can have the same combination of values for all their attributes. You can also ensure that a specific (set of) attribute(s) (instead of the combination of the whole record) within an entity must be unique (i.e. social security number, order id, etc). Any such set of attributes is called a **super key**. A superkey SK specifies a **uniqueness constraint** that no two distinct tuples in any state r of R can have the same value for SK (however, some of the fields can have repeated values):

super key {social security number, name, age} (name and age can exist already, the set of these 3 fields is still unique)

Every relation has at least one default superkey— the set of all its attributes.

A more handy tool is to provide a **key**:

1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This uniqueness property also applies to a superkey.
2. It is a **minimal super key** that is, if you remove (part of) it, you could potentially lose uniqueness, whereas in a superkey you dont need to be minimal (that is, you can have reduncancy).

A key is a superkey but not vice versa.

Any superkey formed from a single attribute is also a key.

A key with multiple attributes must require all its attributes together to have the uniqueness property

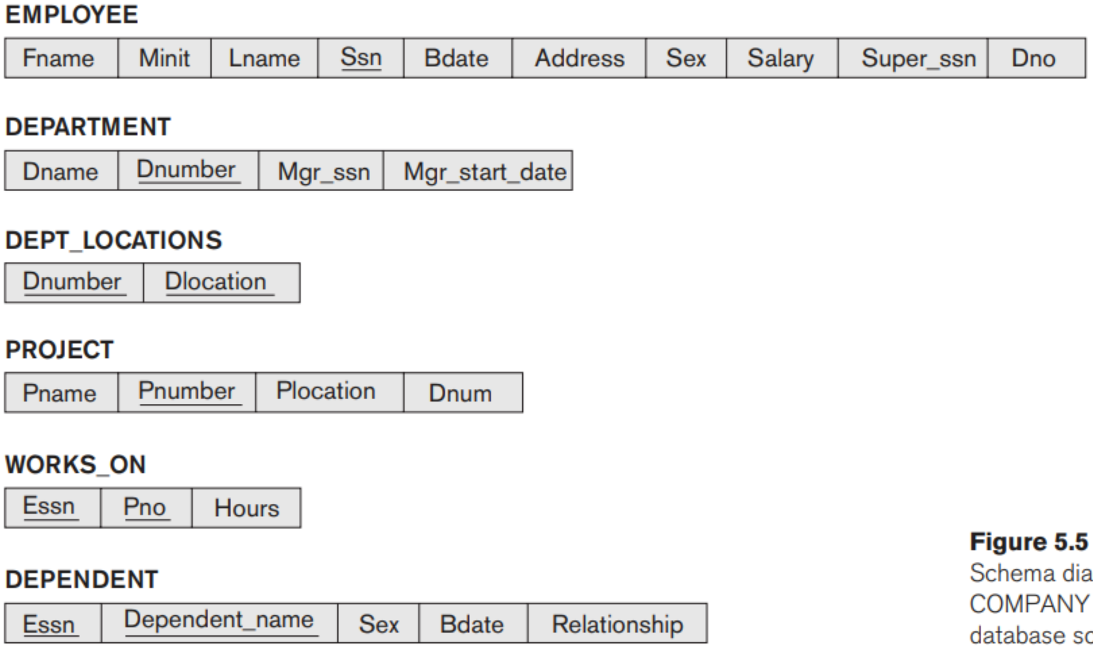
A key is time-invariant It must continue to hold when we insert new tuples in the relation.

More than one unique fields in a relation can be unique, we would could have then **primary key** and **candidate key**, the choice can be arbitrary but both have the "uniqueness" constraint. Usually the primary key has a smaller and easier to sort set of attributes.]

NOT NULL is a common constrain, especially among keys, who must be unique.

5.2.3 Relational Databases and Relational Database Schemas

A relational database schema S is a set of relation schemas S = {R_1, R_2, … , R_m} and a set of integrity constraints IC. That is a set of Tables (that often represent an entity) related with each other under a "schema" S. I.e (shopping cart (id, time, quantity, customer), product properties(description, price, product, isAvaiable))



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS

Dnumber	Dlocation

PROJECT

Pname	Pnumber	Plocation	Dnum

WORKS_ON

Essn	Pno	Hours

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

Figure 5.5
Schema diagram for the COMPANY relational database schema.

5.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

Underline fields are primary keys, when there are 2 underlined fields is because one of them is a **foreign key** that is, a pointer to a primary key at another table that contains the attributes assigned to such id.

The **entity integrity constraint** states that **no primary key value can be NULL**.

The **referential integrity** constraints means that a foregin key must match an existing primary key somewhere.

Figure 5.7
Referential integrity constraints displayed on the COMPANY relational database schema.

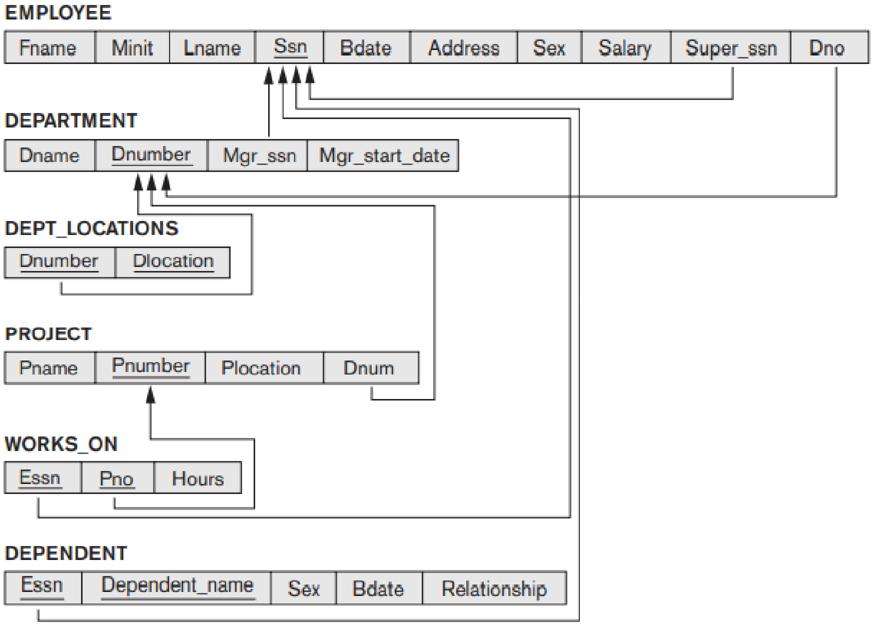
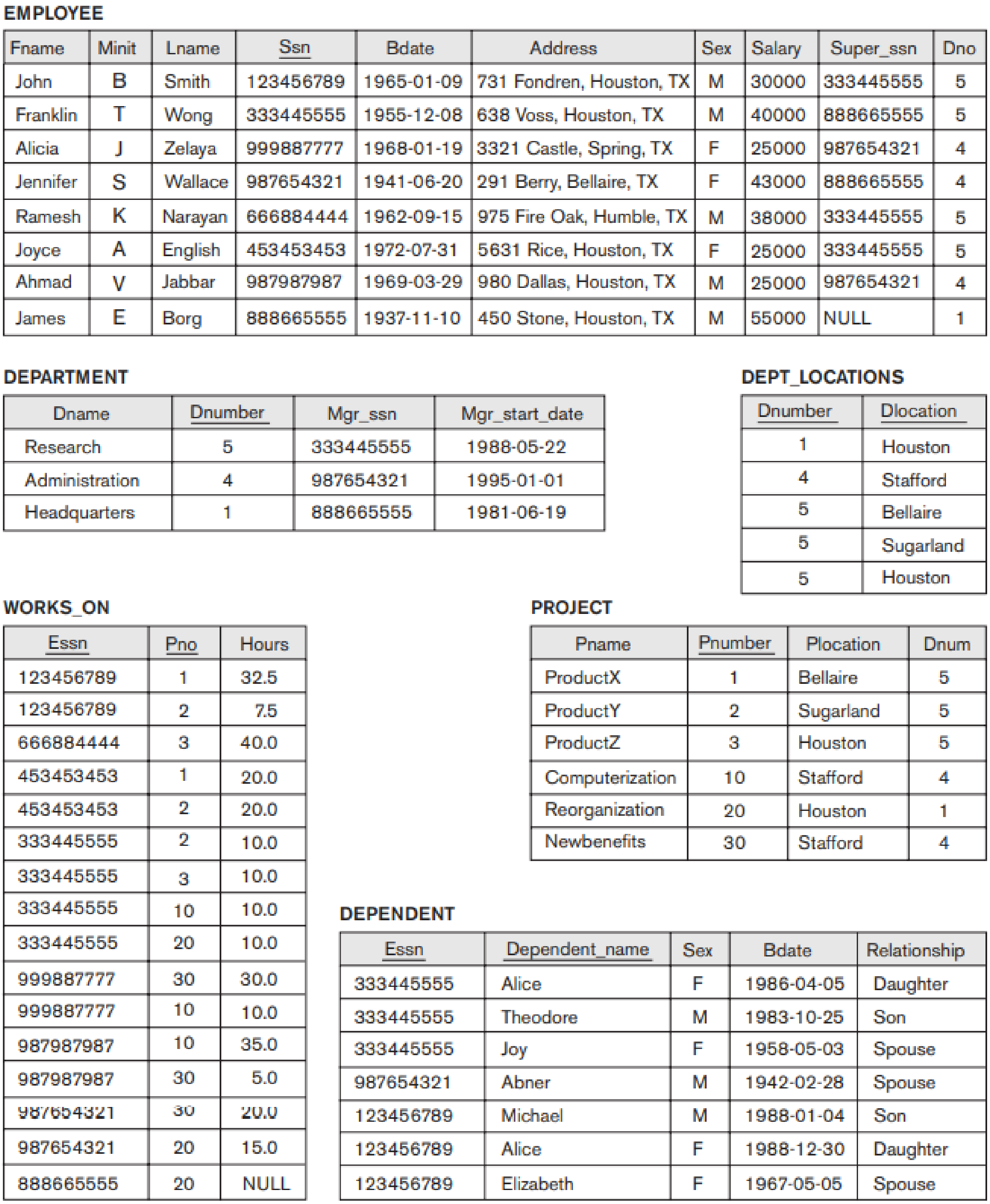


Figure 5.6
One possible database state for the COMPANY relational database schema.



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

In SQL, the CREATE TABLE statement of the SQL DDL allows the definition of primary key, unique key, NOT NULL, entity integrity, and referential integrity constraints, among other constraints.

5.3 Insert, Delete, and Update Operations dealing with constraints

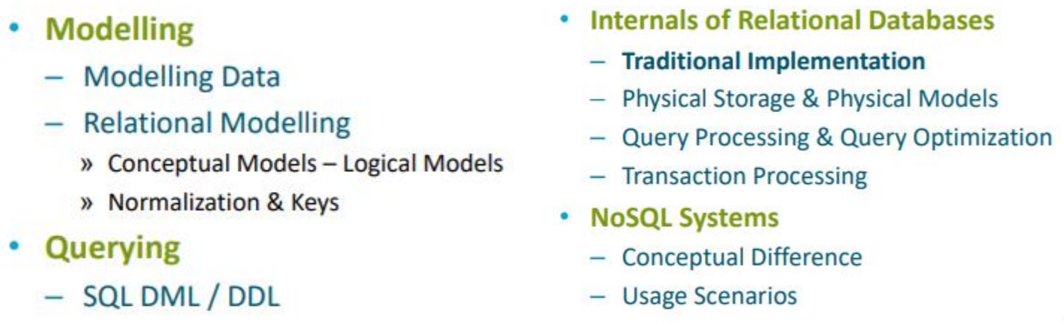
- If an insertion violates one or more constraints, the default option is to reject the insertion.
- The Delete operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database. Several options are available if a deletion operation causes a violation. The first option, called restrict, is to reject the deletion. The second option, called cascade, is to attempt to cascade (or propagate) the deletion by deleting tuples that reference the tuple that is being deleted. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with Essn = ‘999887777’. A third option, called set null or set default, is to modify the referencing attribute values that cause the violation; each such value is either set to NULL or changed to a reference value. The default option can be specifed by the DDL of the DBMS
- Updating an attribute that is neither part of a primary key nor part of a foreign key usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain. Just make sure that PK remains unique and check out for dependencies with foreign keys. The same options as with delete apply, most commonly reject or cascade.

5. 4 The Transaction Concept

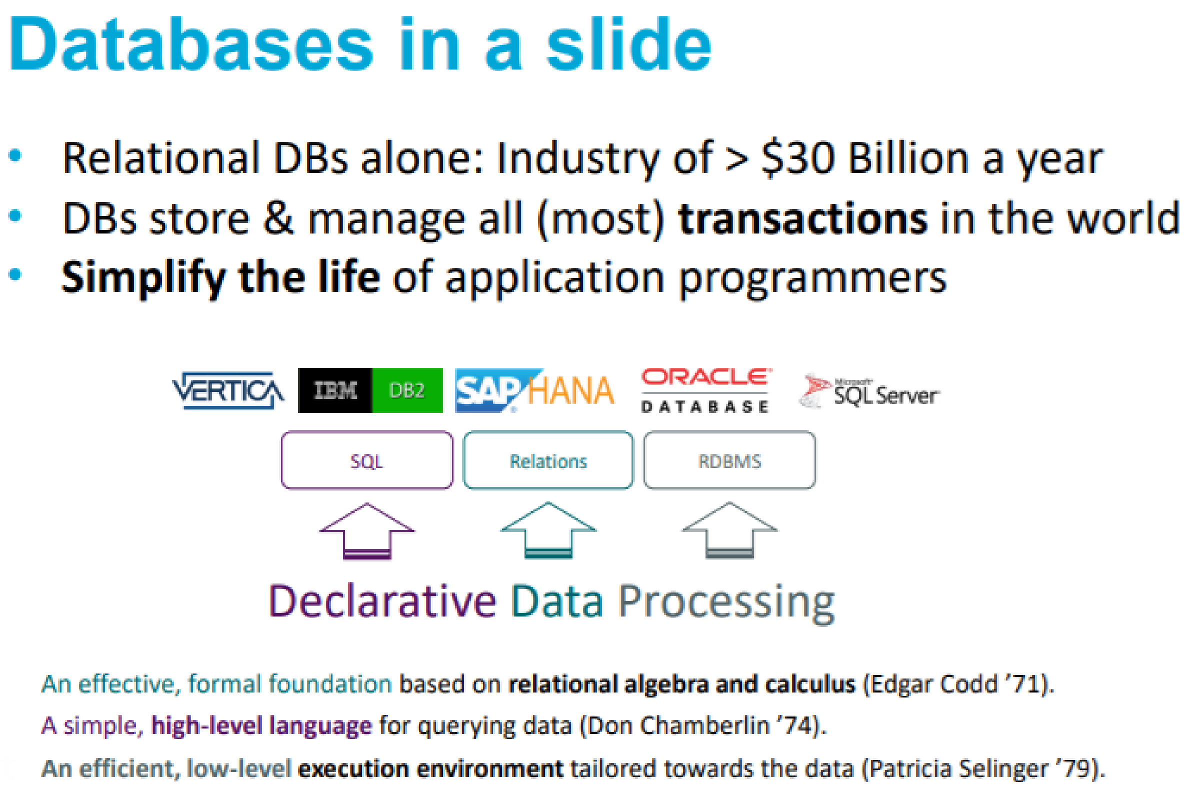
A transaction is an executing program that includes some **database operations that form together an atomic unit of work** against the database. Their either **completed as a whole or rejected all**. A large number of commercial applications running against relational databases in online **transaction processing (OLTP)** systems are executing transactions at rates that reach several hundred per second.

Lecture 1 Introduction to Database Systems

Focus of the course:



- Modelling
  - Modelling Data
  - Relational Modelling
    - » Conceptual Models – Logical Models
    - » Normalization & Keys
- Querying
  - SQL DML / DDL
- Internals of Relational Databases
  - **Traditional Implementation**
  - Physical Storage & Physical Models
  - Query Processing & Query Optimization
  - Transaction Processing
- NoSQL Systems
  - Conceptual Difference
  - Usage Scenarios



Databases in a slide

- Relational DBs alone: Industry of > $30 Billion a year
- DBs store & manage all (most) **transactions** in the world
- **Simplify the life** of application programmers

VERTICA
IBM DB2
SAP HANA
ORACLE DATABASE
Microsoft SQL Server
SQL
Relations
RDBMS

Declarative Data Processing

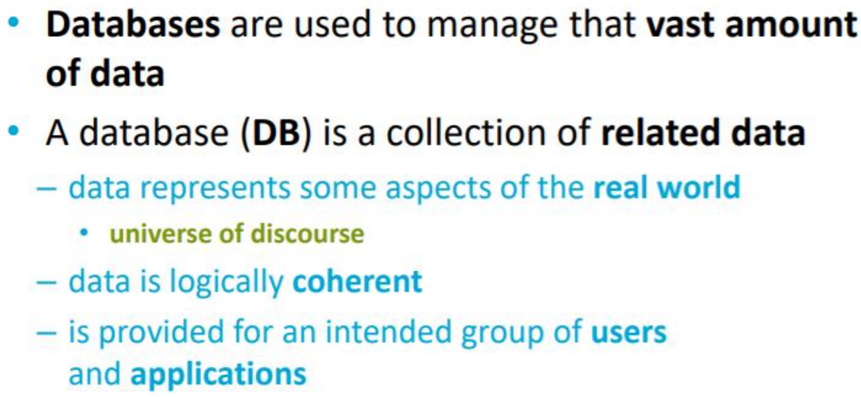
An effective, formal foundation based on **relational algebra and calculus** (Edgar Codd ’71).

A simple, **high-level language** for querying data (Don Chamberlin ’74).

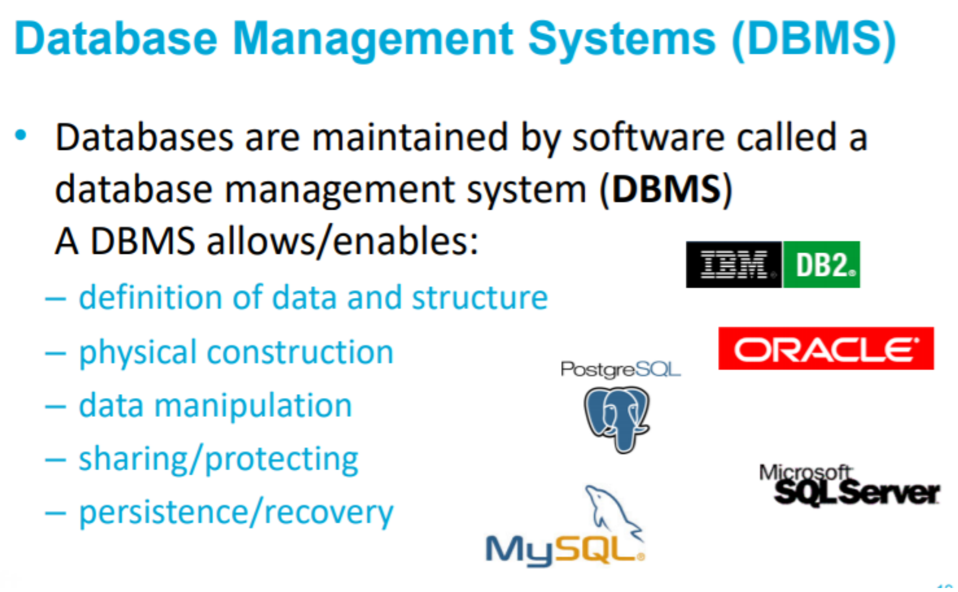
An efficient, low-level **execution environment** tailored towards the data (Patricia Selinger ’79).

The need for Data Management

- **Managing** large amounts of **data** is an integral part of most nowadays business and governmental activities
- Databases developed successfully since the 1960s
  - One of the most successful technologies in computers science!

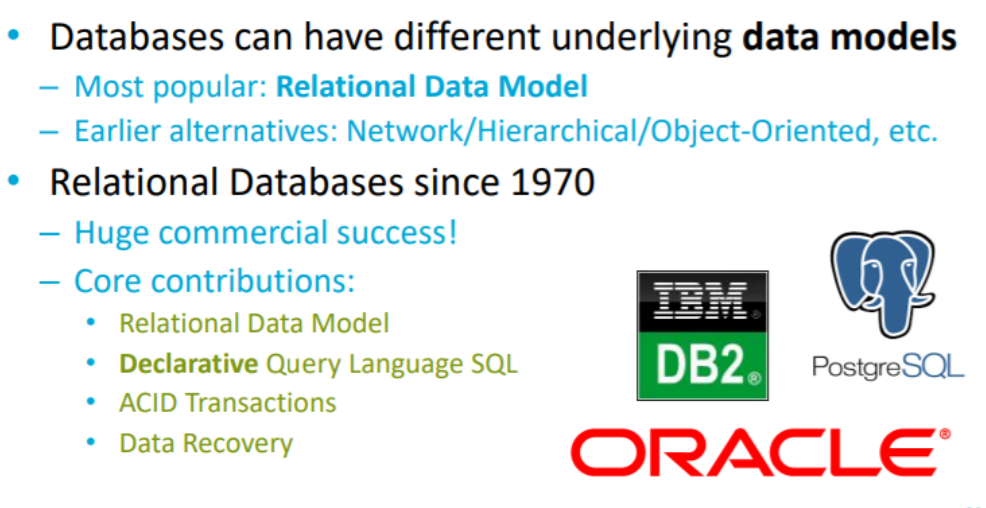


- **Databases** are used to manage that **vast amount of data**
- A database (**DB**) is a collection of **related data**
  - data represents some aspects of the **real world**
    - universe of discourse
  - data is logically **coherent**
  - is provided for an intended group of **users** and **applications**

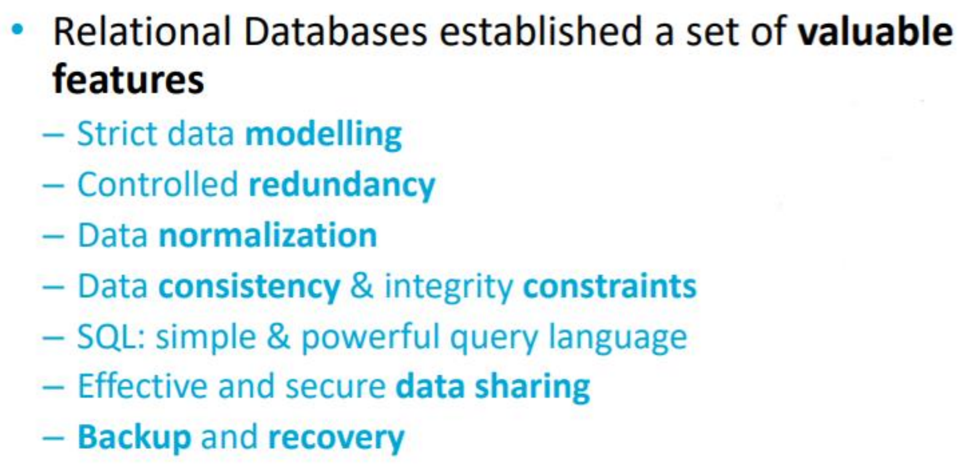


Database Management Systems (DBMS)

- Databases are maintained by software called a database management system (**DBMS**)
  A DBMS allows/enables:
  - definition of data and structure
  - physical construction
  - data manipulation
  - sharing/protecting
  - persistence/recovery



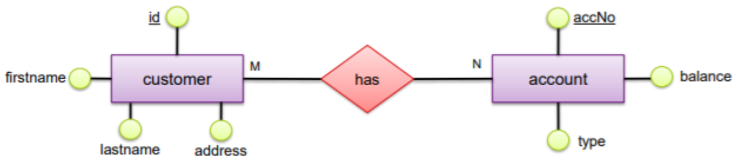
- Databases can have different underlying **data models**
  - Most popular: **Relational Data Model**
  - Earlier alternatives: Network/Hierarchical/Object-Oriented, etc.
- Relational Databases since 1970
  - Huge commercial success!
  - Core contributions:
    - Relational Data Model
    - **Declarative** Query Language SQL
    - ACID Transactions
    - Data Recovery



- Relational Databases established a set of **valuable features**
  - Strict data **modelling**
  - Controlled **redundancy**
  - Data **normalization**
  - Data **consistency** & integrity **constraints**
  - SQL: simple & powerful query language
  - Effective and secure **data sharing**
  - **Backup** and **recovery**

(relational)

- Databases are **well-structured** (e.g. ER-Model)
  - **Catalog** (data dictionary) contains all **meta-data**
  - Defines the **structure** of the data in the database
- Example: ER-Model
  - Simple banking system (Conceptual Model)



- Resulting tables (logical "model")

Customer

ID	firstname	lastname	address
1	Aard	Vark	123AB
2	Bandi	Coot	231CX
3	Cham	Eleon	12dXX

Customer2Account

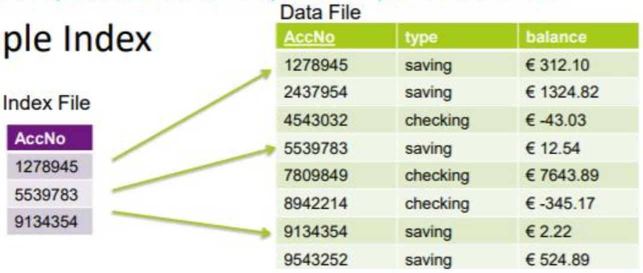
ID	accNo
1	10
1	11
2	12

Account

accNo	type	balance
10	Check	0
11	Saving	232
12	Check	-232

Characteristics of Relational Databases

- Databases aim at efficient manipulation of data
  - Physical tuning allows for good data allocation
  - Indexes speed up search and access
  - Query plans are optimized for improved performance
- Example: Simple Index



Index File

AccNo
1278945
5539783
9134354

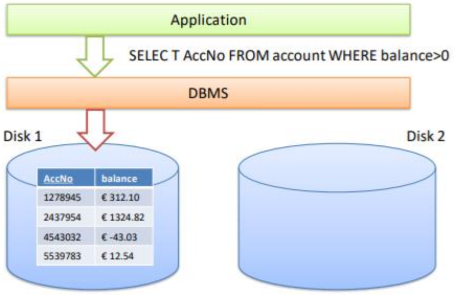
Data File

AccNo	type	balance
1278945	saving	€ 312.10
2437954	saving	€ 1324.82
4543032	checking	€ -43.03
5539783	saving	€ 12.54
7809849	checking	€ 7643.89
8942214	checking	€ -345.17
9134354	saving	€ 2.22
9543252	saving	€ 524.89

Data Independence

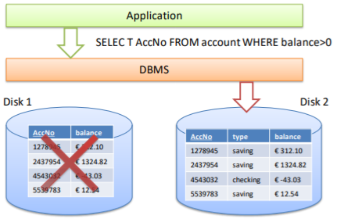
- Independence of applications and data
  - Databases employ data abstraction by providing **data models**
  - Applications work only on the conceptual representation of data
    - Data is strictly typed (Integer, Timestamp, VarChar,...)
    - Details on where data is actually stored and how it is accessed is hidden by the DBMS
    - Applications can access and manipulate data by invoking declarative operations (e.g. SQL Select statements)
  - DBMS-controlled parts of the file system are strongly protected against outside manipulation

- **Example:** Schema is changed and table-space moved without an application noticing



AccNo	balance
1278945	€ 312.10
2437954	€ 1324.82
4543032	€ -43.03
5539783	€ 12.54

- **Example:** Schema is changed and table-space moved without an application noticing



AccNo	balance
1278945	€ 312.10
2437954	€ 1324.82
4543032	€ -43.03
5539783	€ 12.54

AccNo	type	balance
1278945	saving	€ 312.10
2437954	saving	€ 1324.82
4543032	checking	€ -43.03
5539783	saving	€ 12.54

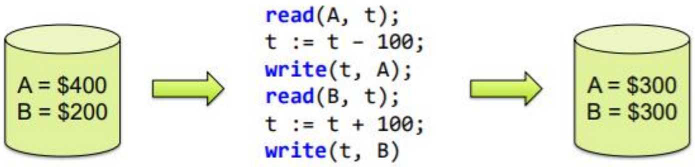
Views on the data

- Supports multiple **views** of the data
  - Views provide a different perspective of the DB
    - A user's conceptual understanding or task-based excerpt of all data (e.g. aggregations)
    - Security considerations and access control (e.g. projections)
  - For the application, a view does not differ from a table
  - Views may contain **subsets** of a DB and/or contain **virtual data**
    - Virtual data is **derived** from the DB (mostly by simple SQL statements, e.g. joins over several tables)
    - Can either be computed at query time or **materialized** upfront

Transactions: concurrently accessing data

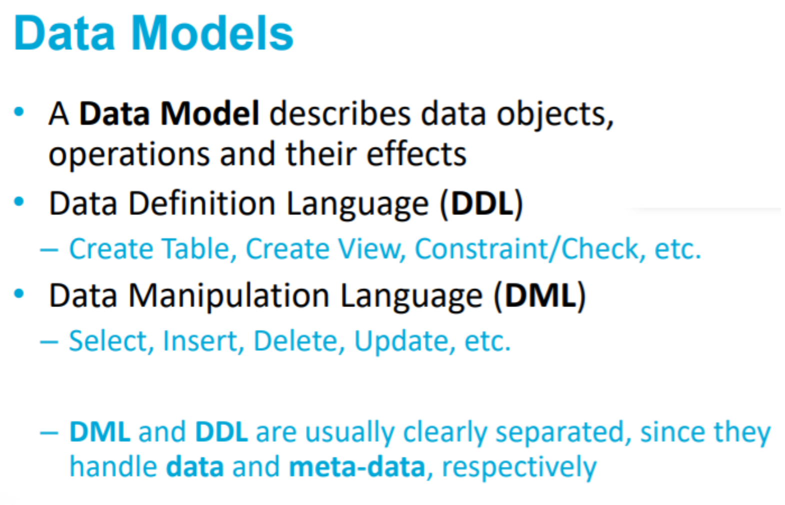
- **Sharing** of data and support for **atomic multi-user** transactions
  - Multiple user and applications may access the DB at the same time
  - **Concurrency control** is necessary for maintaining consistency

- A transaction is a unit of work, possibly containing multiple data accesses and updates, that must commit or abort as a single unit, and follows the ACID principles
- e.g, transfer $100 from account A to account B
- Transactions introduce two problems:
  - Recovery: What if the system fails in the middle of execution?
  - Concurrency: What happens when two transactions try to access the same object?



```
read(A, t);
t := t - 100;
write(t, A);
read(B, t);
t := t + 100;
write(t, B)
```

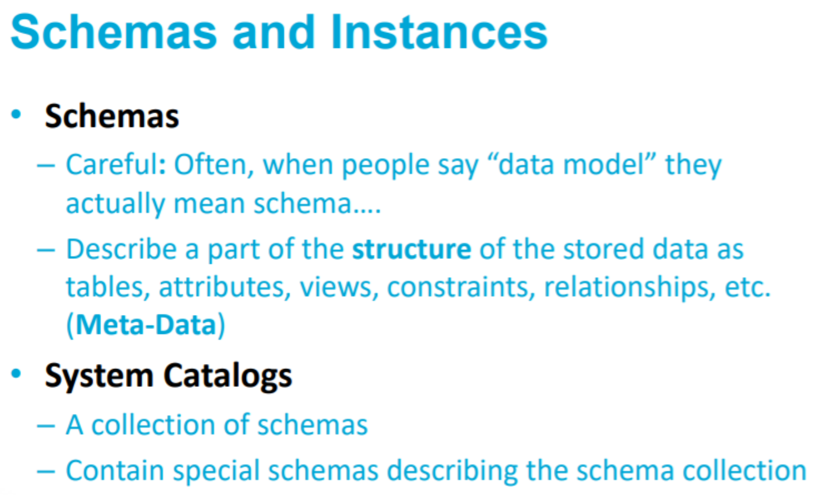
- **A**tomicity
  - A transaction is either executed completely (commit) **or** not at all (abort)
- **C**onsistency
  - A transation transforms a consistent database state into a (possibly different) consistent database state
- **I**solation
  - A transaction is executed in isolation (i.e., does not see any effect of other concurrently running („uncommitted") transactions).
- **D**urability
  - A successfully completed („committed") transaction has a permanent effect on the database



Data Models

- A **Data Model** describes data objects, operations and their effects
- Data Definition Language (**DDL**)
  - Create Table, Create View, Constraint/Check, etc.
- Data Manipulation Language (**DML**)
  - Select, Insert, Delete, Update, etc.

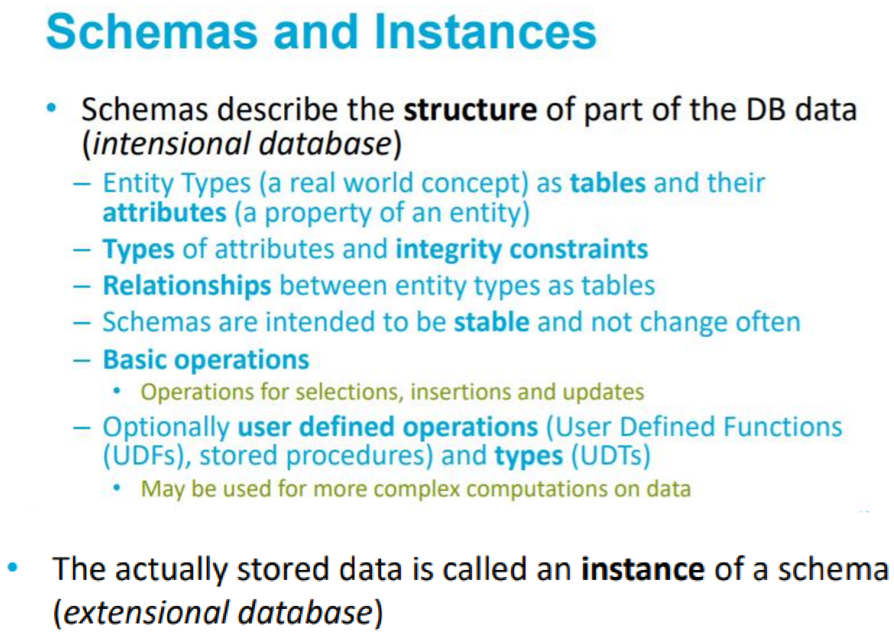
  - **DML** and **DDL** are usually clearly separated, since they handle **data** and **meta-data**, respectively



Schemas and Instances

- **Schemas**
  - Careful**:** Often, when people say “data model” they actually mean schema….
  - Describe a part of the **structure** of the stored data as tables, attributes, views, constraints, relationships, etc. (**Meta-Data**)
- **System Catalogs**
  - A collection of schemas
  - Contain special schemas describing the schema collection

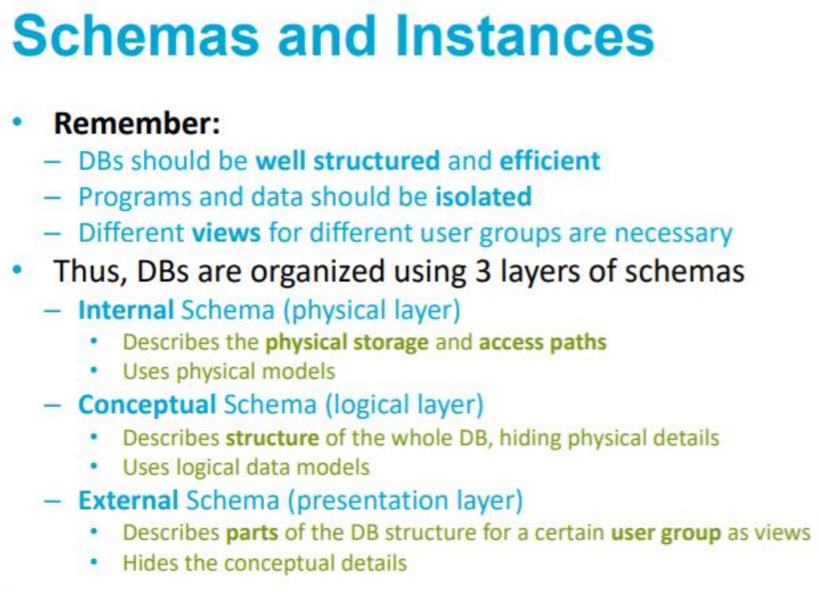
(Schema example: all the relationships between tables, views and table properties of my food app)



Schemas and Instances

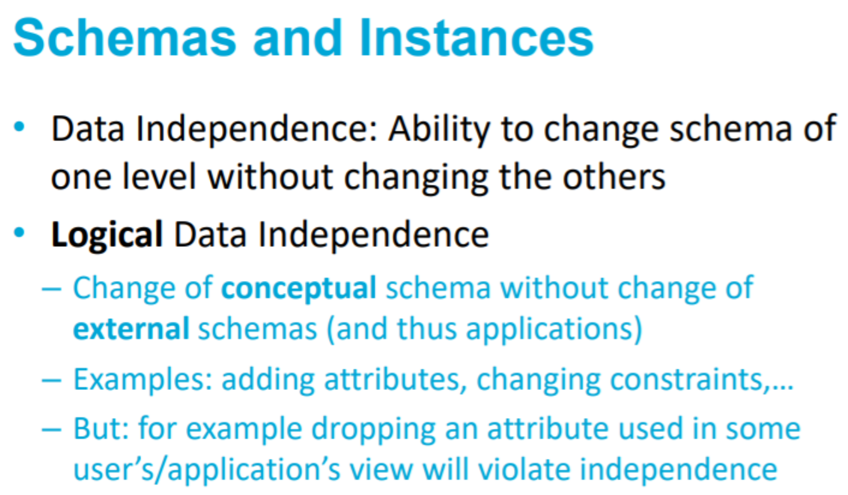
- Schemas describe the **structure** of part of the DB data (*intensional database*)
  - Entity Types (a real world concept) as **tables** and their **attributes** (a property of an entity)
  - **Types** of attributes and **integrity constraints**
  - **Relationships** between entity types as tables
  - Schemas are intended to be **stable** and not change often
  - **Basic operations**
    - Operations for selections, insertions and updates
  - Optionally **user defined operations** (User Defined Functions (UDFs), stored procedures) and **types** (UDTs)
    - May be used for more complex computations on data

- The actually stored data is called an **instance** of a schema (*extensional database*)



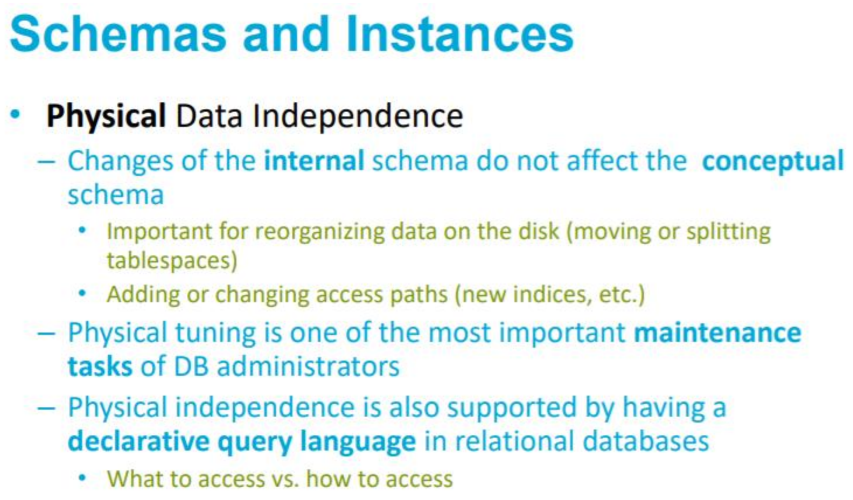
Schemas and Instances

- **Remember:**
  - DBs should be **well structured** and **efficient**
  - Programs and data should be **isolated**
  - Different **views** for different user groups are necessary
- Thus, DBs are organized using 3 layers of schemas
  - **Internal** Schema (physical layer)
    - Describes the **physical storage** and **access paths**
    - Uses physical models
  - **Conceptual** Schema (logical layer)
    - Describes **structure** of the whole DB, hiding physical details
    - Uses logical data models
  - **External** Schema (presentation layer)
    - Describes **parts** of the DB structure for a certain **user group** as views
    - Hides the conceptual details



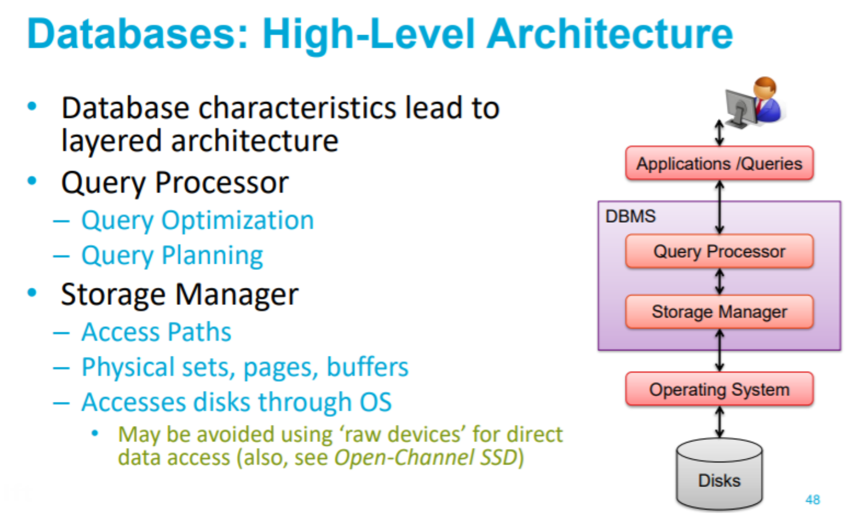
Schemas and Instances

- Data Independence: Ability to change schema of one level without changing the others
- **Logical** Data Independence
  - Change of **conceptual** schema without change of **external** schemas (and thus applications)
  - Examples: adding attributes, changing constraints,…
  - But: for example dropping an attribute used in some user's/application's view will violate independence



Schemas and Instances

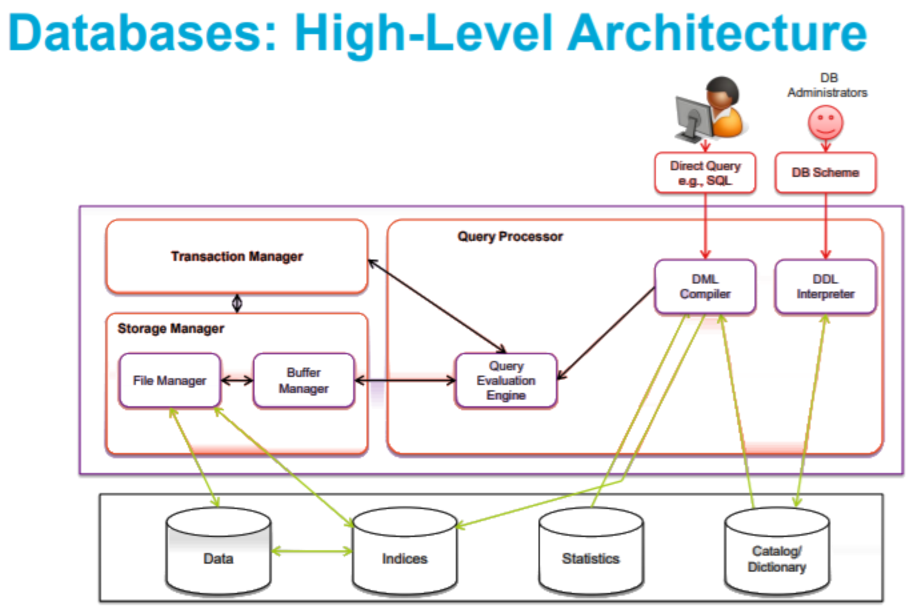
- **Physical** Data Independence
  - Changes of the **internal** schema do not affect the **conceptual** schema
    - Important for reorganizing data on the disk (moving or splitting tablespaces)
    - Adding or changing access paths (new indices, etc.)
  - Physical tuning is one of the most important **maintenance tasks** of DB administrators
  - Physical independence is also supported by having a **declarative query language** in relational databases
    - What to access vs. how to access



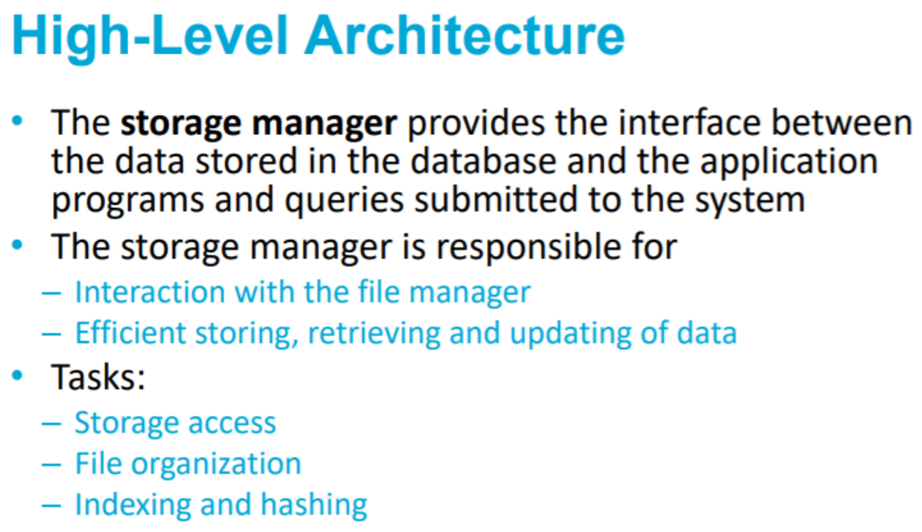
Databases: High-Level Architecture

- Database characteristics lead to layered architecture
- Query Processor
  - Query Optimization
  - Query Planning
- Storage Manager
  - Access Paths
  - Physical sets, pages, buffers
  - Accesses disks through OS
    - May be avoided using 'raw devices' for direct data access (also, see *Open-Channel SSD*)

48

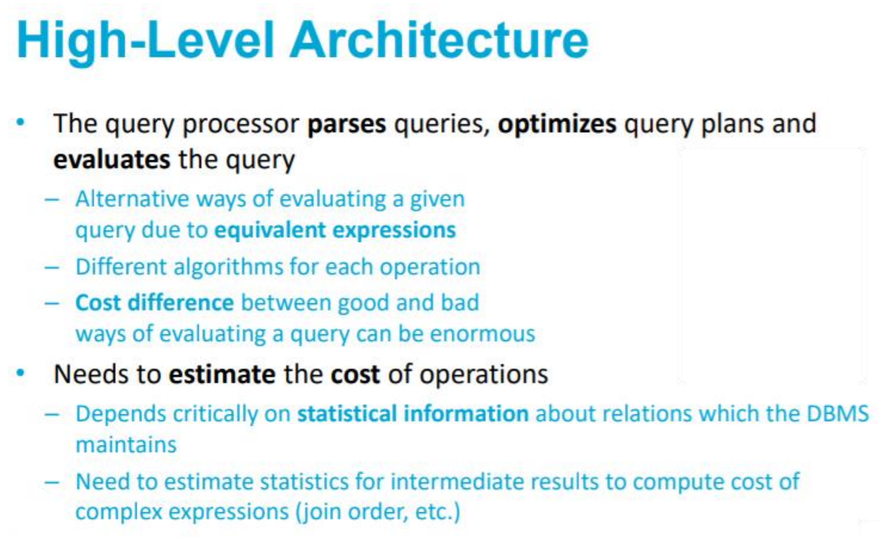


Databases: High-Level Architecture



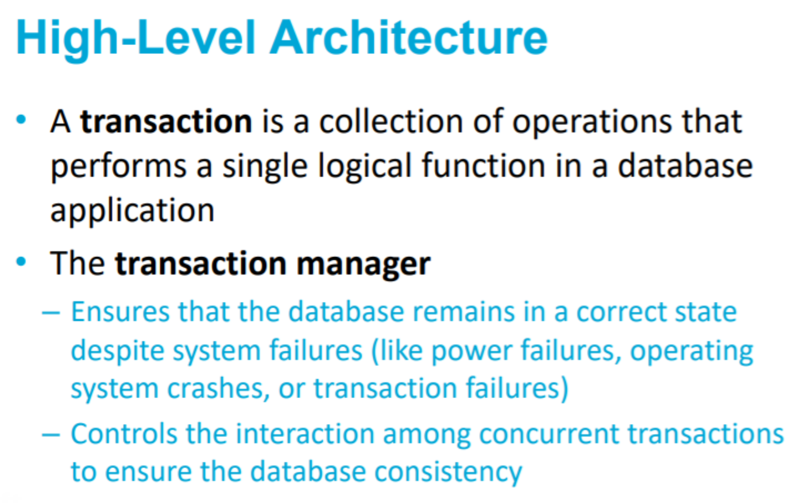
High-Level Architecture

- The **storage manager** provides the interface between the data stored in the database and the application programs and queries submitted to the system
- The storage manager is responsible for
  - Interaction with the file manager
  - Efficient storing, retrieving and updating of data
- Tasks:
  - Storage access
  - File organization
  - Indexing and hashing



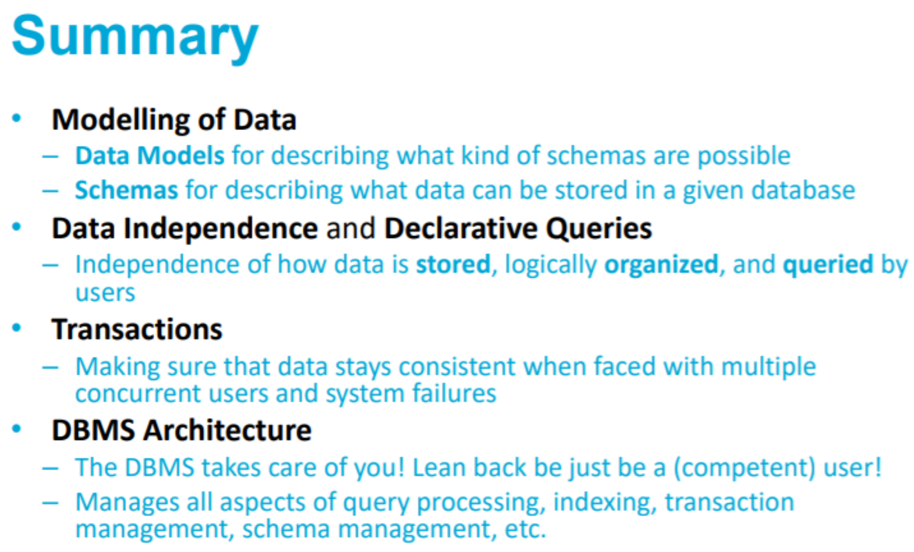
High-Level Architecture

- The query processor **parses** queries, **optimizes** query plans and **evaluates** the query
  - Alternative ways of evaluating a given query due to **equivalent expressions**
  - Different algorithms for each operation
  - **Cost difference** between good and bad ways of evaluating a query can be enormous
- Needs to **estimate** the **cost** of operations
  - Depends critically on **statistical information** about relations which the DBMS maintains
  - Need to estimate statistics for intermediate results to compute cost of complex expressions (join order, etc.)



High-Level Architecture

- A **transaction** is a collection of operations that performs a single logical function in a database application
- The **transaction manager**
  - Ensures that the database remains in a correct state despite system failures (like power failures, operating system crashes, or transaction failures)
  - Controls the interaction among concurrent transactions to ensure the database consistency



Summary

- **Modelling of Data**
  - **Data Models** for describing what kind of schemas are possible
  - **Schemas** for describing what data can be stored in a given database
- **Data Independence** and **Declarative Queries**
  - Independence of how data is **stored**, logically **organized**, and **queried** by users
- **Transactions**
  - Making sure that data stays consistent when faced with multiple concurrent users and system failures
- **DBMS Architecture**
  - The DBMS takes care of you! Lean back be just be a (competent) user!
  - Manages all aspects of query processing, indexing, transaction management, schema management, etc.

Chapter 3. Data Modeling with Entity-Relationship (ER) Model

The entity-relationship model is a popular high-level conceptual data model used for the conceptual design of database applications.

Unified Modeling Language is an object modeling methodology that go beyond tabase design to specify detailed design of software modules and their interactions using various types of diagrams. UML's class diagrams are very similar to ER diagrams.

3.1 Using High-Level Conceptual Data Models for Database Design

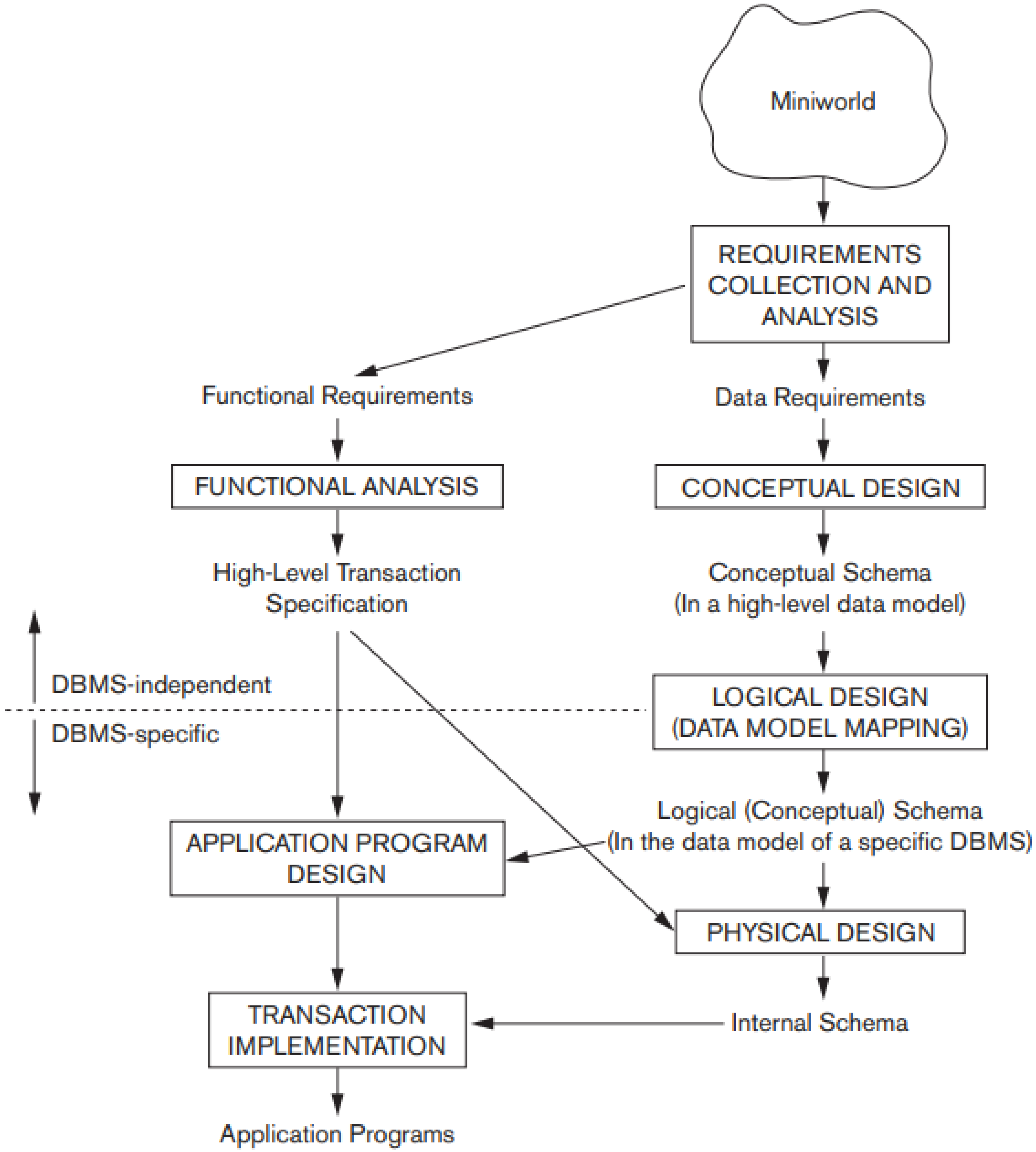

Miniworld
REQUIREMENTS COLLECTION AND ANALYSIS
Functional Requirements
Data Requirements
FUNCTIONAL ANALYSIS
CONCEPTUAL DESIGN
High-Level Transaction Specification
Conceptual Schema (In a high-level data model)
DBMS-independent
DBMS-specific
LOGICAL DESIGN (DATA MODEL MAPPING)
Logical (Conceptual) Schema (In the data model of a specific DBMS)
APPLICATION PROGRAM DESIGN
PHYSICAL DESIGN
TRANSACTION IMPLEMENTATION
Internal Schema
Application Programs

Figure 3.1
A simplified diagram to illustrate the main phases of database design.

- **Mini world:** the part of the real world that will be represented in the database.
- **Requirements collections and analysis:** The database designers interview prospective database users to understand and document their data requirements
- **Functional requirements:** User defined operations (or transactions) that will be applied to the database, including both retrievals and updates. This is part of Software Engineering.
- **Conceptual schema:** is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints.
- **logical design or data model mapping:** implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational (SQL) model—so the conceptual schema is transformed from the high-level data model into the implementation data model.
- **physical design:** internal storage structures, file organizations, indexes, access paths, and physical design parameters for the database files are specified.

Conceptual Design (Schema)

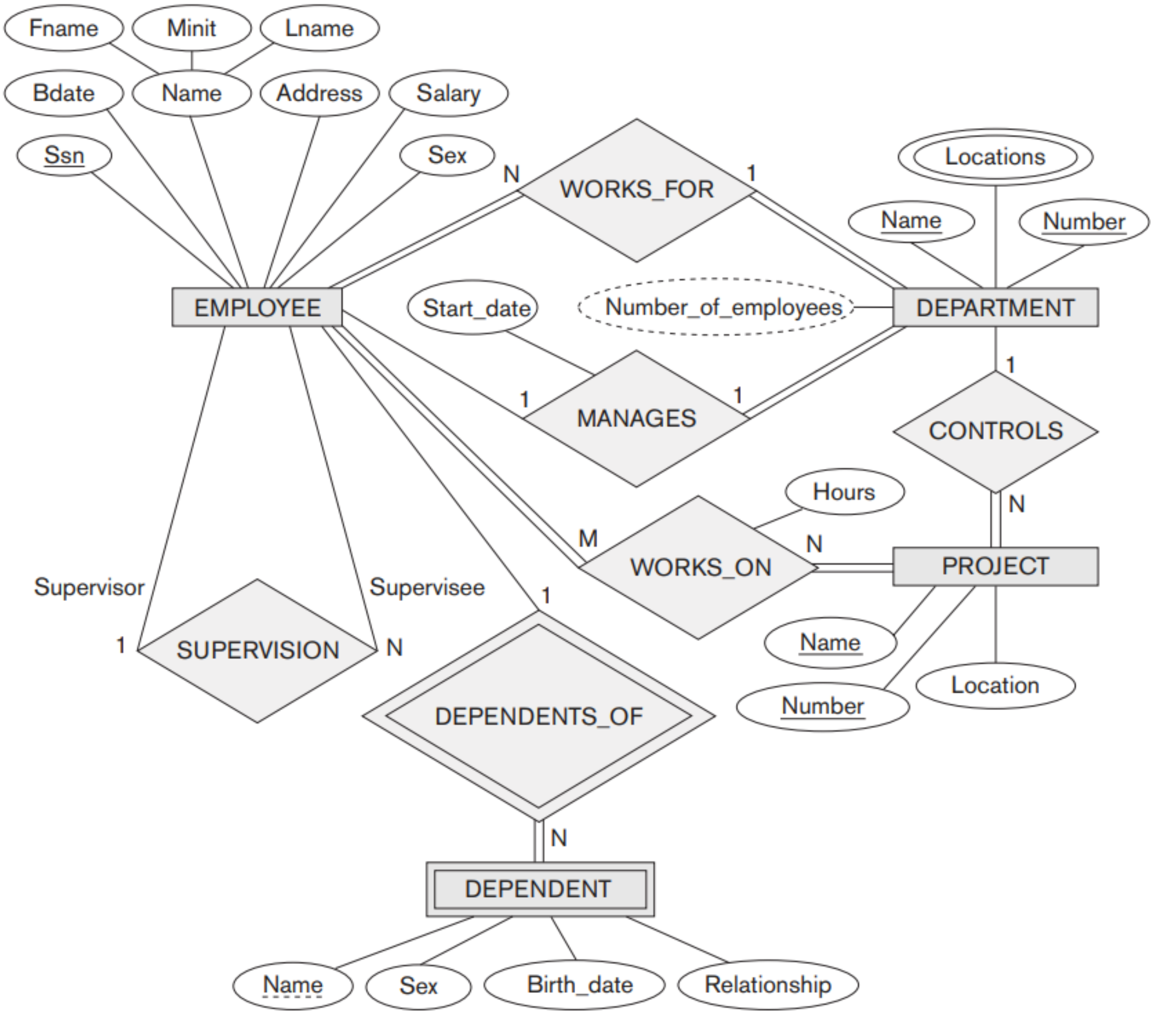


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter and is summarized in Figure 3.14.

3.3 Entity Types, Entity Sets, Attributes and Keys

The basic concept that the ER model represents is an entity, which is a thing or object in the real world with an independent existence. Entities ‘i.e. employee x has attributes (ssn, bank account, email, name, phone number, position, works_for, manages)

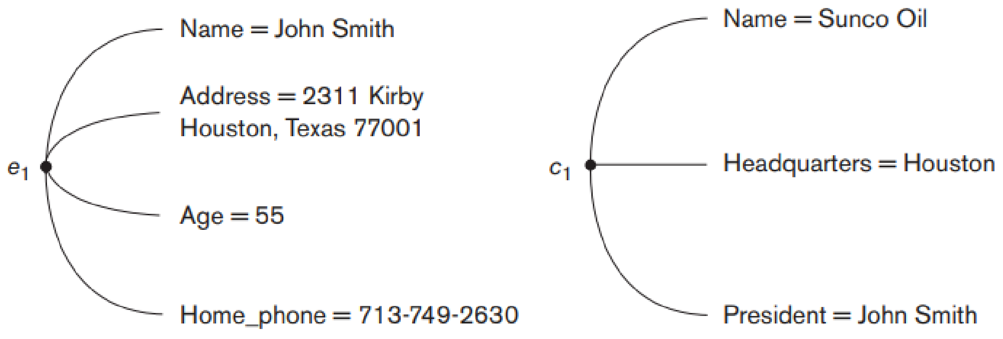


Figure 3.3
Two entities, EMPLOYEE e_1, and COMPANY c_1, and their attributes.

Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the EMPLOYEE entity shown in Figure 3.3 can be subdivided into Street_address, City, State, and Zip, 3 with the values ‘2311 Kirby’, ‘Houston’, ‘Texas’, and ‘77001’. Attributes that are not divisible are called simple or atomic attributes.

The value of a composite attribute is the concatenation of the values of its component simple attributes.

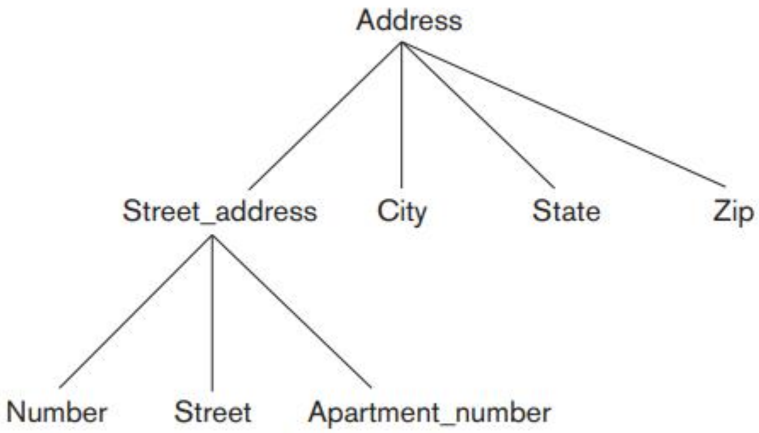
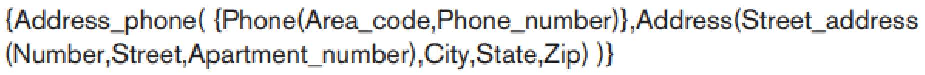


Figure 3.4
A hierarchy of composite attributes.

Single-value vs multivalued: Some attributes can also have an optional number of values such as “college degree”, where the acceptable values would consist of NULL, degree 1, degree 2..., degree n. There can be an upperbound and lowerbound for n.

Stored vs derived: Some values you store, such as Birth_date, others are computed automatically, such as Age, these are derived.

Complex attributes: combination of composite (‘{‘ for taxonomy) , and multivalued (comma separated).:



{Address_phone({Phone(Area_code,Phone_number)},Address(Street_address (Number,Street,Apartment_number),City,State,Zip))}

Figure 3.5
A complex attribute: Address_phone.

Entity type

Entity Type is Like a Class, i.e. ‘Student’, where entities are like objects, i.e. var s = new Student(); where s is an instance of Student. The Entity Type and the Entity Type Set are two separate concepts, but they same word i.e. EMPLOYEE may be used to refeer to both these concept. While the Entity Type is like a Java class and defines the properties of its Set in the DBMS schema, tbe Entity (Type) Set refers to the collection (set) of all the records that are instance of (entities) of such Enitity Type.

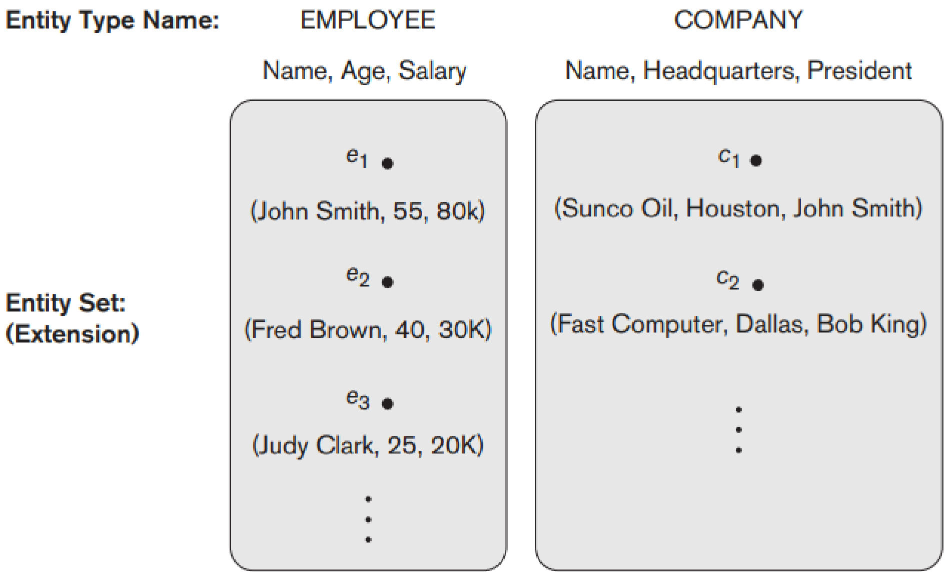
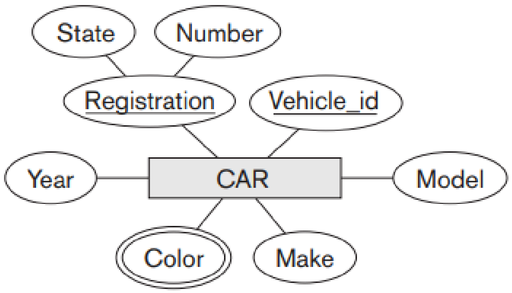


Figure 3.6
Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

An entity type is represented in ER diagrams5 (see Figure 3.2) as a rectangular box enclosing the entity type name. Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals.

(foreign) **key** and/or **uniqueness constraint** are underlined. An entity type **without a key** is called a **weak entity type**. It still has a **parial key** if an attribute happens to be unique, or in the worst case scenario it still has a partial key if you take all of the attributes combined.



An **entity type** describes the **schema or intension** for a set of entities that share the same structure. The collection of entities of a particular entity type is grouped into an **entity set**, which is also called the **extension of the entity type**.

Example

1. An entity type DEPARTMENT with attributes Name, Number, Locations, Manager, and Manager_start_date. Locations is the only multivalued attribute. We can specify that both Name and Number are (separate) key attributes because each was specified to be unique.
2. An entity type PROJECT with attributes Name, Number, Location, and Controlling_department. Both Name and Number are (separate) key attributes.
3. An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary, Birth_date, Department, and Supervisor. Both Name and Address may be composite attributes; however, this was not specified in the requirements. We must go back to the users to see if any of them will refer to the individual components of Name—First_name, Middle_initial, Last_name—or of Address. In our example, Name is modeled as a composite attribute, whereas Address is not, presumably after consultation with the users.
4. An entity type DEPENDENT with attributes Employee, Dependent_name, Sex, Birth_date, and Relationship (to the employee).

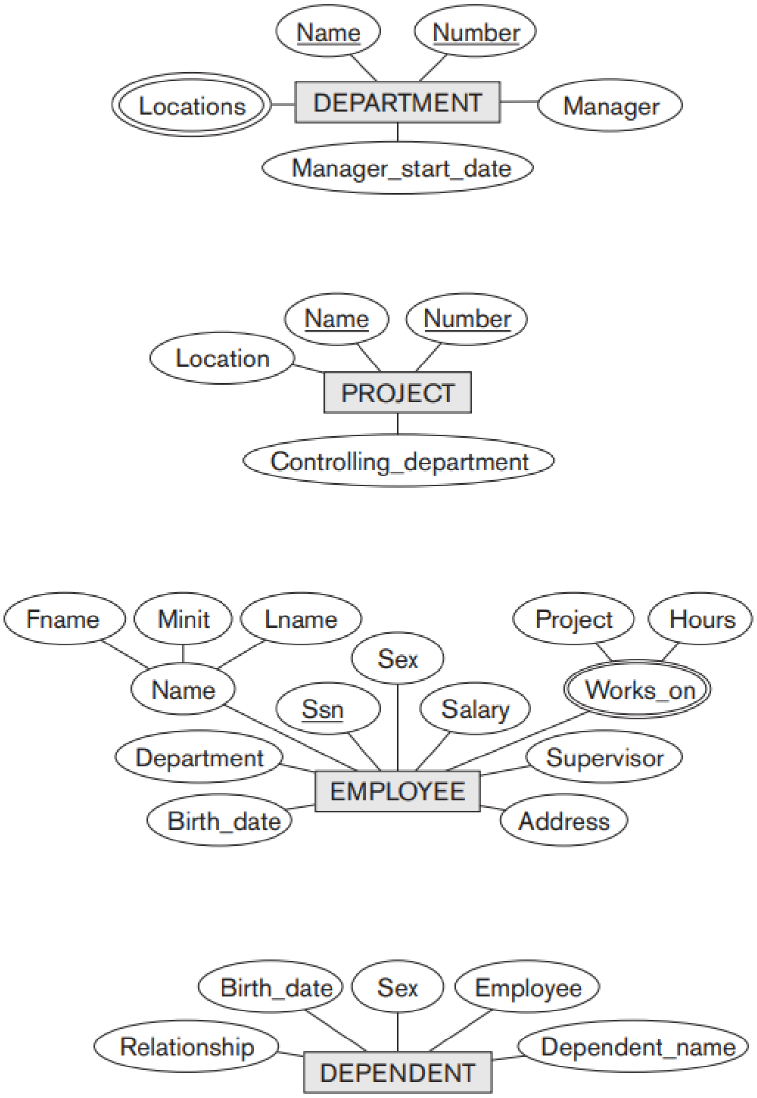


Figure 3.8
Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

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

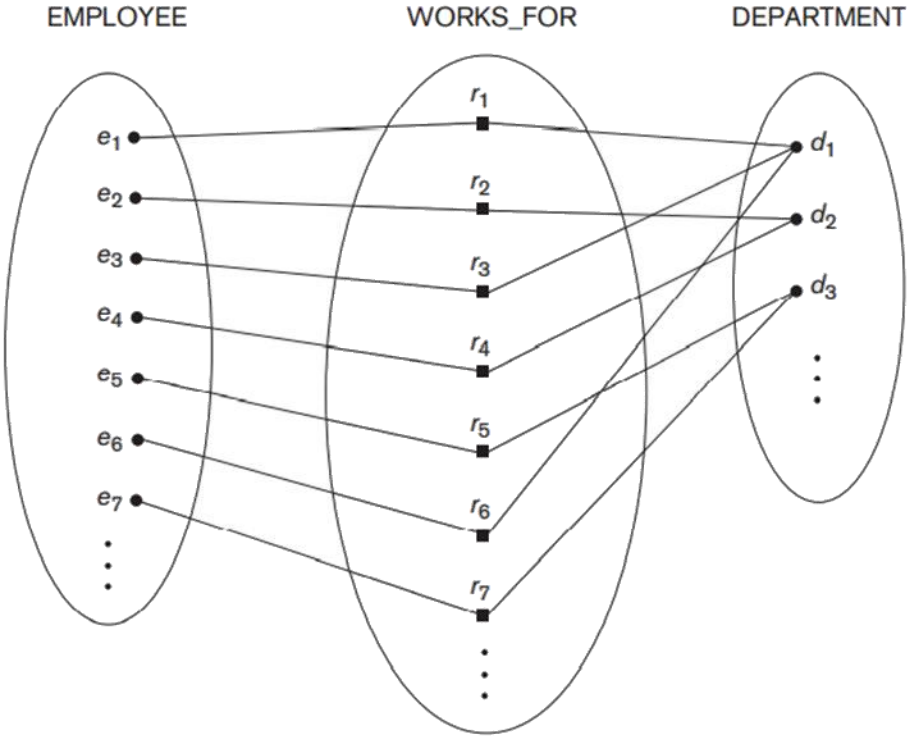


Figure 3.9
Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

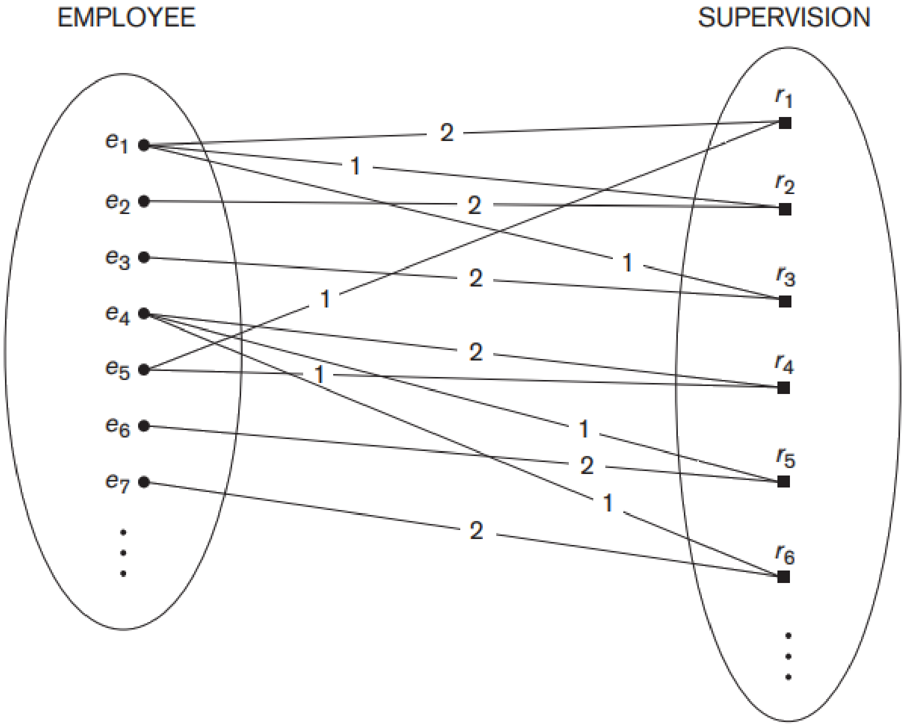


Figure 3.11
A recursive relationship SUPERVISION between EMPLOYEE in the *supervisor* role (1) and EMPLOYEE in the *subordinate* role (2).

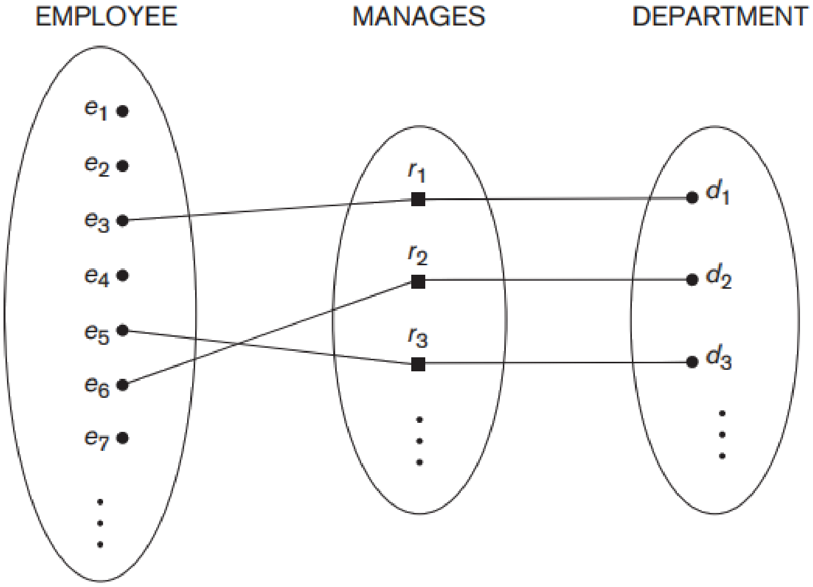


Figure 3.12
A 1:1 relationship, MANAGES.

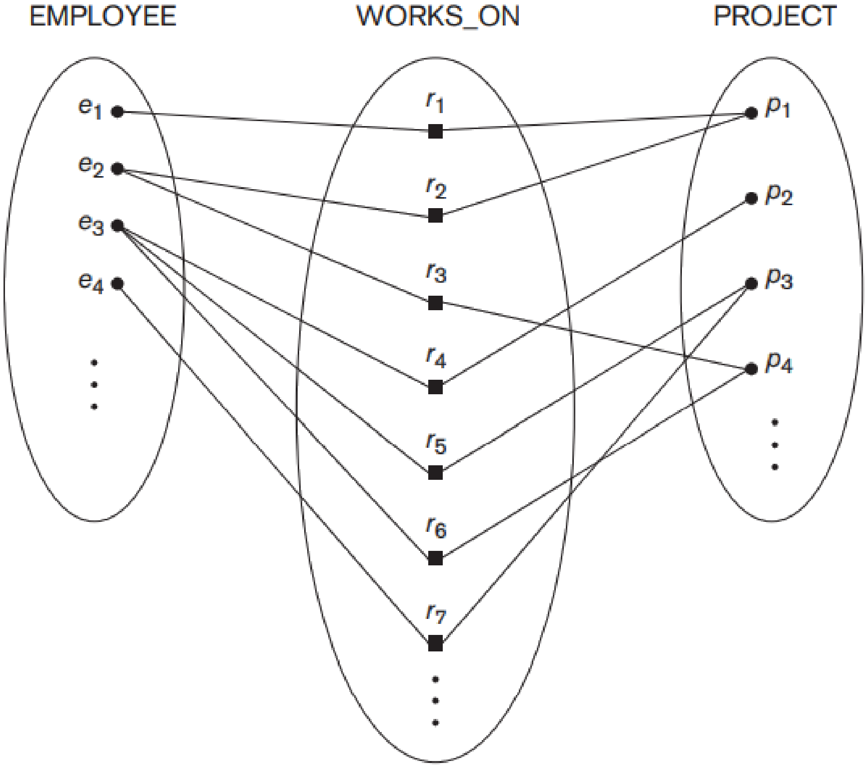
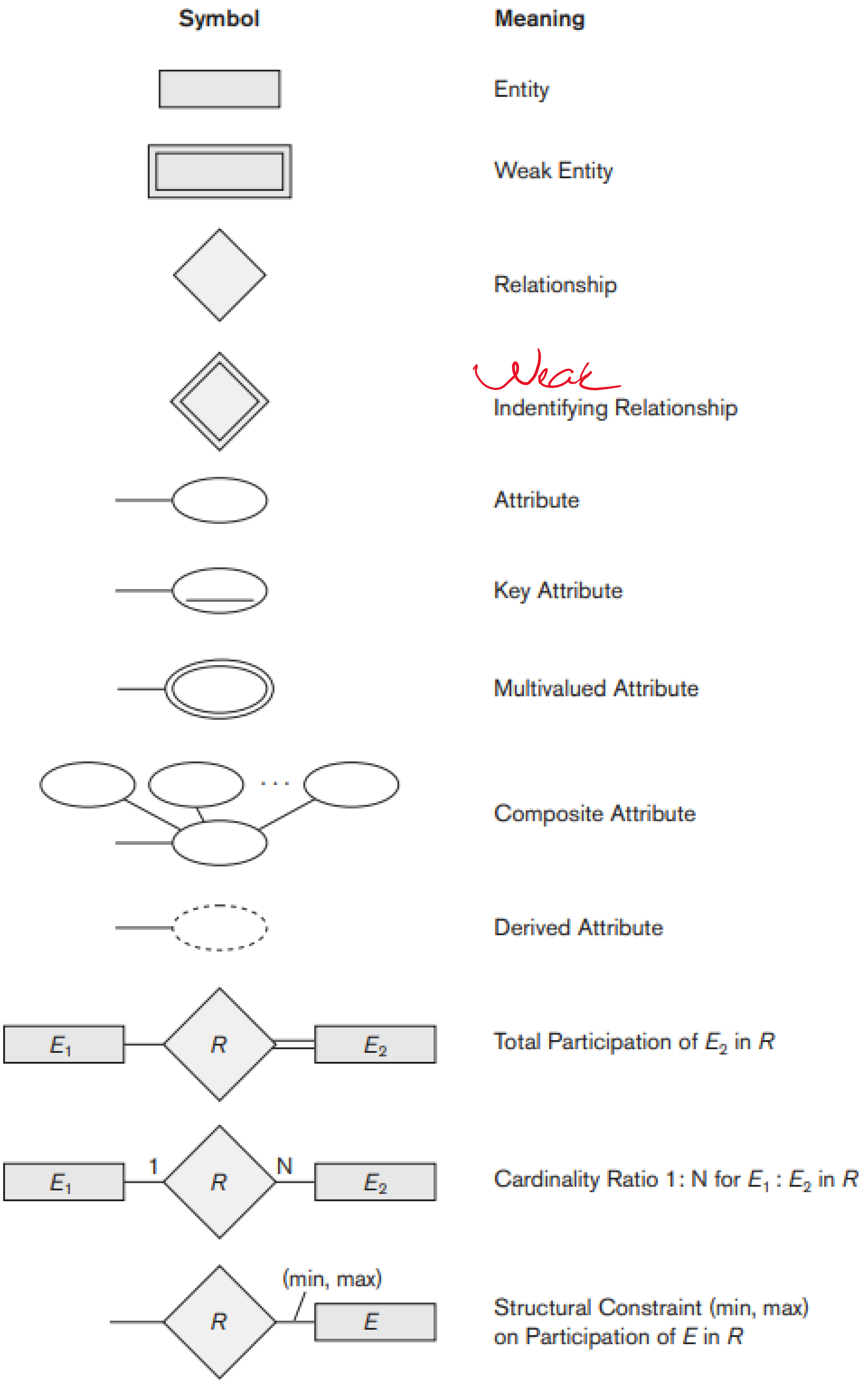


Figure 3.13
An M:N relationship, WORKS_ON.

3.7 ER Diagram Naming Conventions



Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Weak Indentifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
. . .	Composite Attribute
	Derived Attribute
E_1 R E_2	Total Participation of E_2 in R
E_1 1 R N E_2	Cardinality Ratio 1: N for $E_1 : E_2$ in R
R (min, max) E	Structural Constraint (min, max) on Participation of E in R

Figure 3.14
Summary of the notation for ER diagrams.

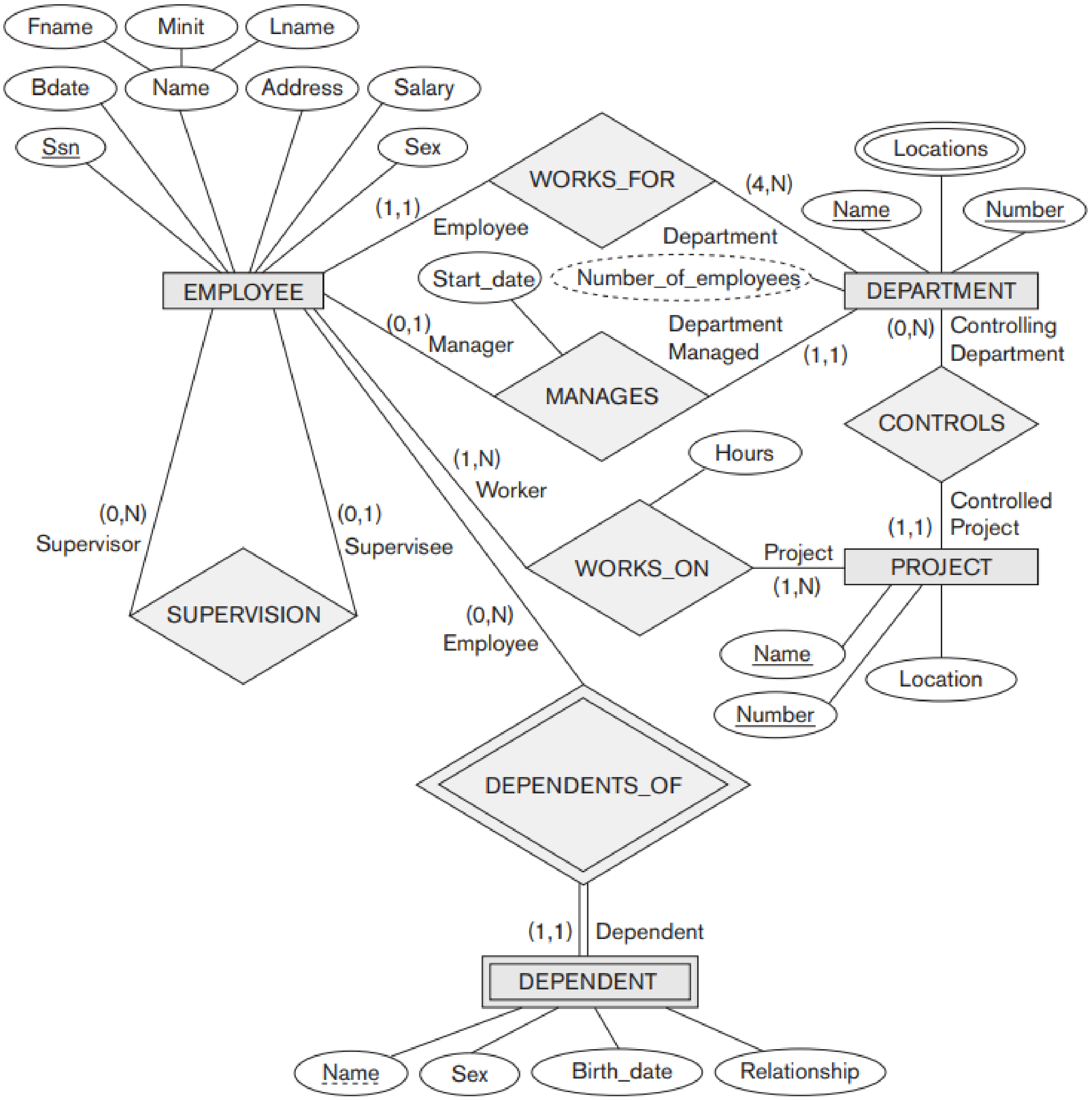
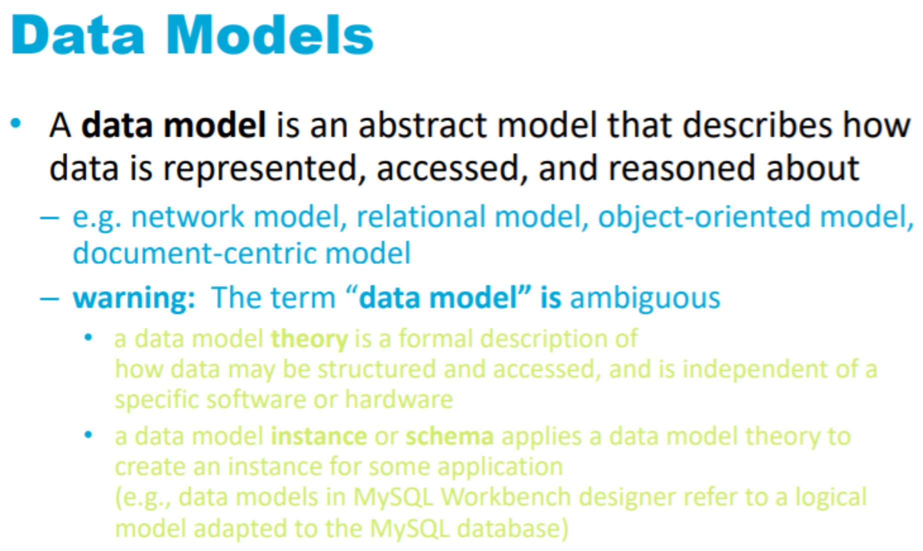


Figure 3.15
ER diagrams for the company schema, with structural constraints specified using (min, max) notation and role names.

In our example, we specify the following relationship types:

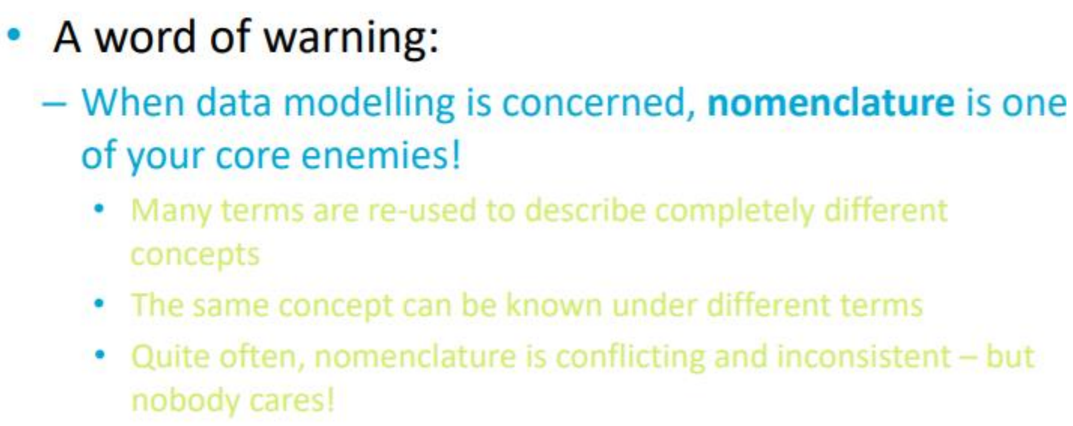
- MANAGES, which is a 1:1(one-to-one) relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is not clear from the requirements. We question the users, who say that a department must have a manager at all times, which implies total participation.[13] The attribute Start_date is assigned to this relationship type.
- WORKS_FOR, a 1:N (one-to-many) relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.
- CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users indicates that some departments may control no projects.
- SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.
- WORKS_ON, determined to be an M:N (many-to-many) relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.
- DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity type DEPENDENT. The participation of EMPLOYEE is partial, whereas that of DEPENDENT is total.

Lecture 2. Introduction to Modelling

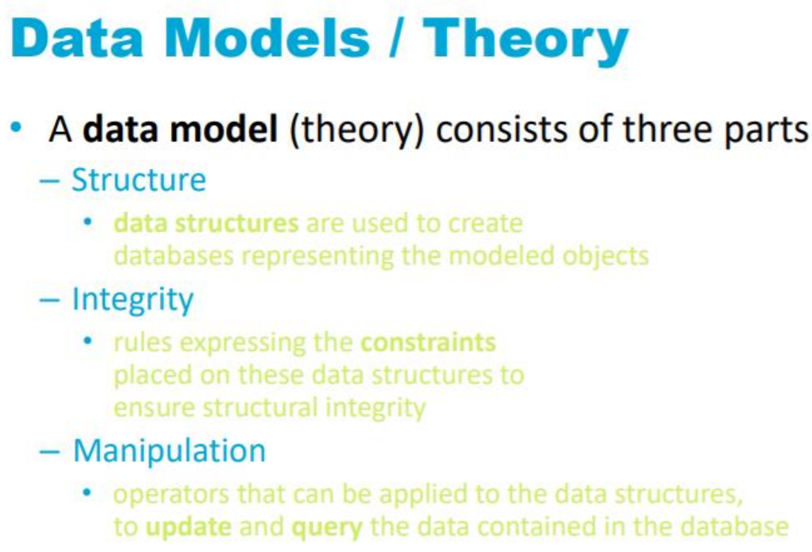


Data Models

- A **data model** is an abstract model that describes how data is represented, accessed, and reasoned about
  - e.g. network model, relational model, object-oriented model, document-centric model
  - **warning:** The term "**data model" is** ambiguous
    - a data model **theory** is a formal description of how data may be structured and accessed, and is independent of a specific software or hardware
    - a data model **instance** or **schema** applies a data model theory to create an instance for some application (e.g., data models in MySQL Workbench designer refer to a logical model adapted to the MySQL database)

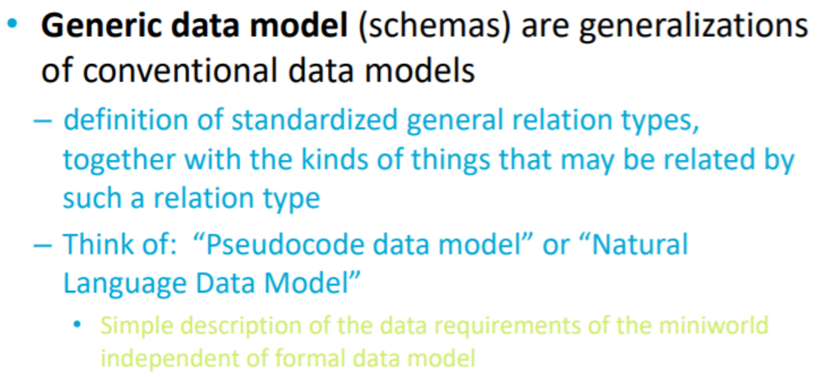


- A word of warning:
  - When data modelling is concerned, **nomenclature** is one of your core enemies!
    - Many terms are re-used to describe completely different concepts
    - The same concept can be known under different terms
    - Quite often, nomenclature is conflicting and inconsistent – but nobody cares!

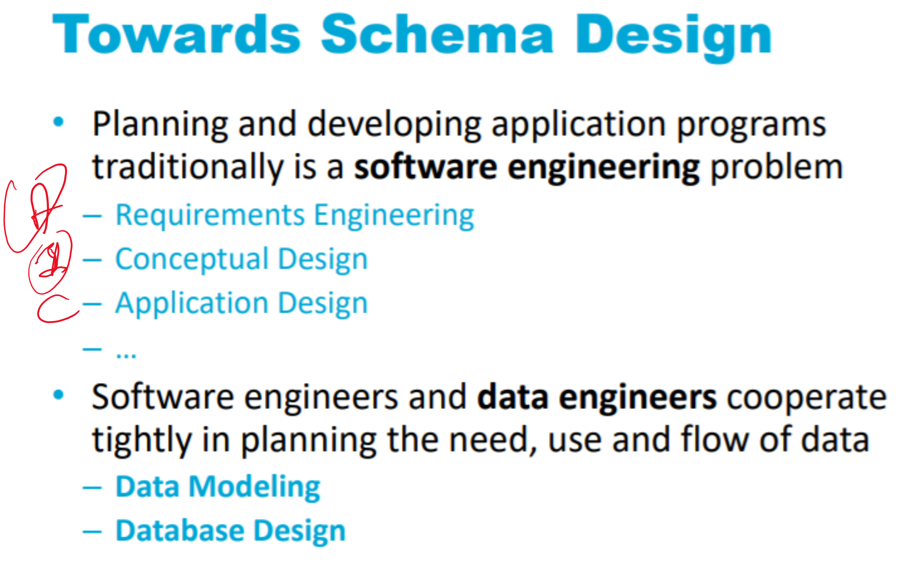


Data Models / Theory

- A **data model** (theory) consists of three parts
  - Structure
    - **data structures** are used to create databases representing the modeled objects
  - Integrity
    - rules expressing the **constraints** placed on these data structures to ensure structural integrity
  - Manipulation
    - operators that can be applied to the data structures, to **update** and **query** the data contained in the database

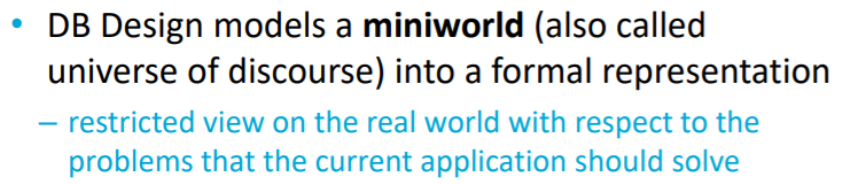


- **Generic data model** (schemas) are generalizations of conventional data models
  - definition of standardized general relation types, together with the kinds of things that may be related by such a relation type
  - Think of: “Pseudocode data model” or “Natural Language Data Model”
    - Simple description of the data requirements of the miniworld independent of formal data model

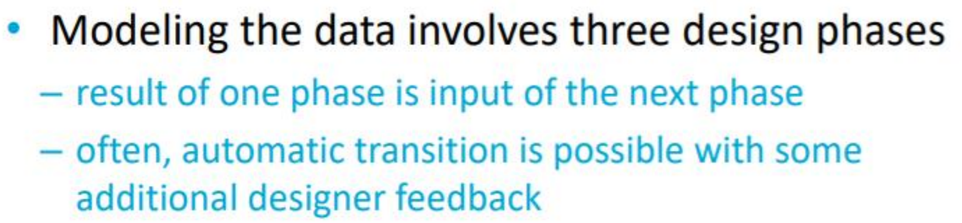


Towards Schema Design

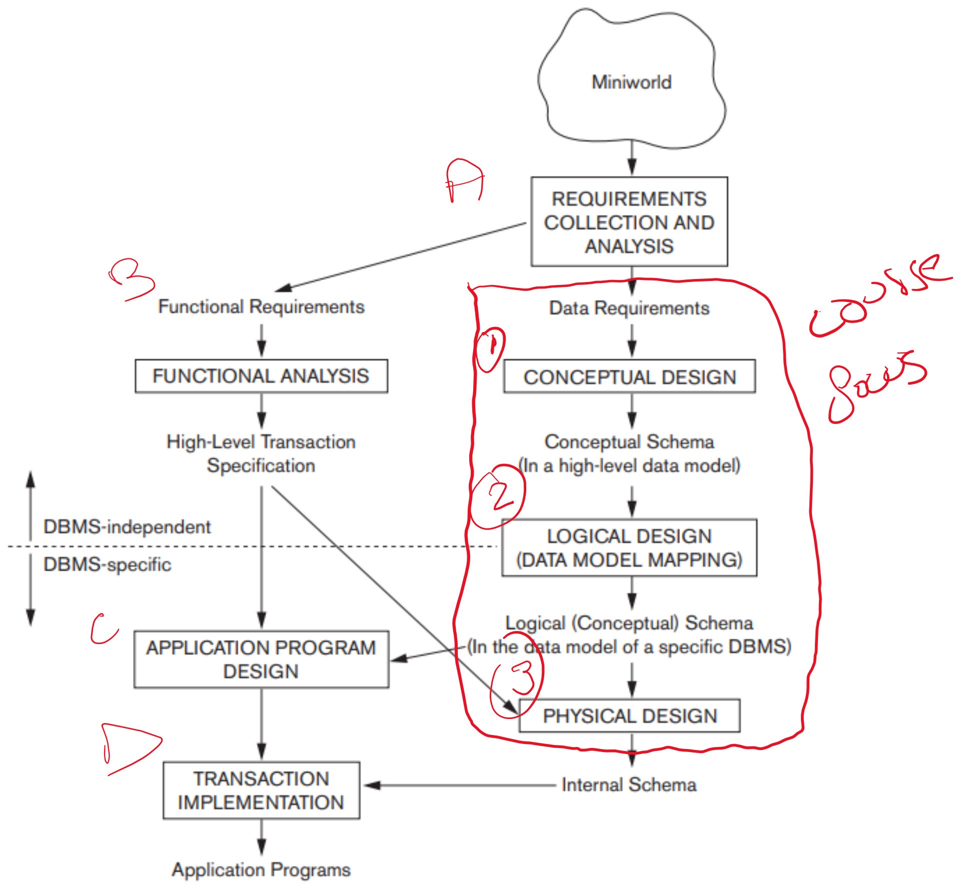
- Planning and developing application programs traditionally is a **software engineering** problem
  - 1 Requirements Engineering
  - 2 Conceptual Design
  - Application Design
  - …
- Software engineers and **data engineers** cooperate tightly in planning the need, use and flow of data
  - **Data Modeling**
  - **Database Design**



- DB Design models a **miniworld** (also called universe of discourse) into a formal representation
  - restricted view on the real world with respect to the problems that the current application should solve



- Modeling the data involves three design phases
  - result of one phase is input of the next phase
  - often, automatic transition is possible with some additional designer feedback


Miniworld
REQUIREMENTS COLLECTION AND ANALYSIS
A
B
Functional Requirements
Data Requirements
course focus
1
FUNCTIONAL ANALYSIS
CONCEPTUAL DESIGN
High-Level Transaction Specification
Conceptual Schema (In a high-level data model)
DBMS-independent
DBMS-specific
2
LOGICAL DESIGN (DATA MODEL MAPPING)
Logical (Conceptual) Schema (In the data model of a specific DBMS)
C
APPLICATION PROGRAM DESIGN
3
PHYSICAL DESIGN
D
TRANSACTION IMPLEMENTATION
Internal Schema
Application Programs

1) - **Conceptual Design**
  - transforms Data Requirements to **conceptual model**
  - describes high-level data entities, relationships, constraints, etc.
    - does not contain any implementation details
    - independent of used software and hardware
    - Only loosely depending on chosen data model

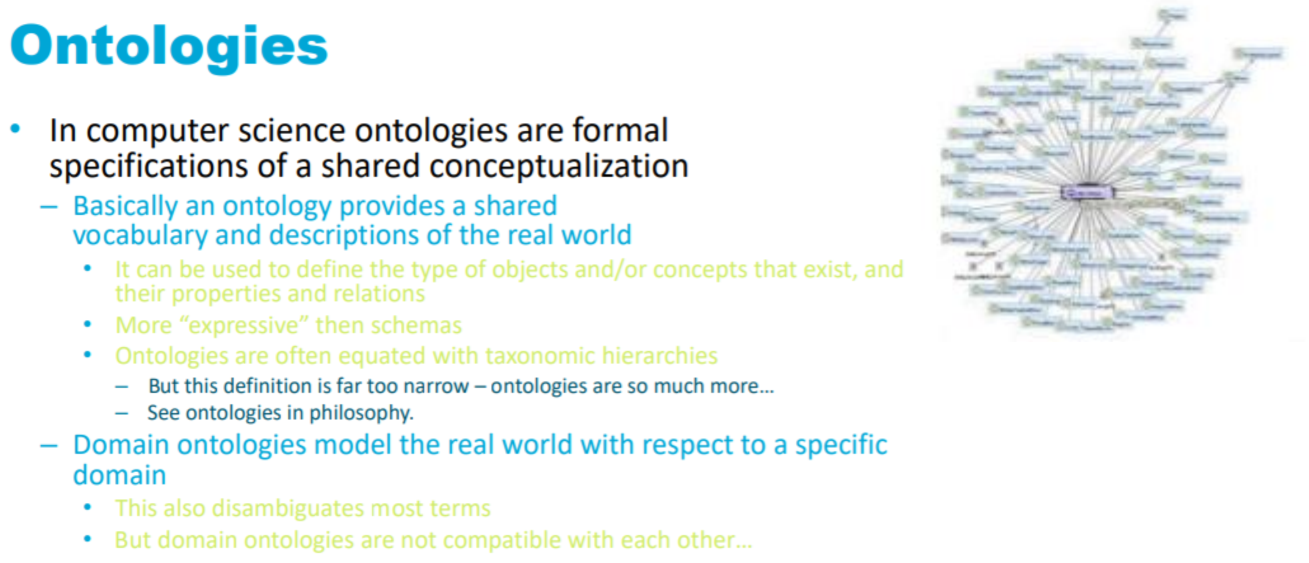
2) - **Logical Design**
  - maps the conceptual data model to the logical data model used by the DBMS
    - e.g. relational model, hierarchical model
    - technology independent conceptual model is adapted to the used DBMS software

3) - **Physical Design**
  - creates internal structures needed to efficiently store/manage data
    - e.g. table spaces, indexes, access paths
    - depends on used hardware and DBMS software

A - **Requirements Analysis**
  - database designers interview prospective **users** and **stakeholders**
  - **Data Requirements** describe what kind of data is needed
  - **Functional Requirements** describe the operations performed on the data

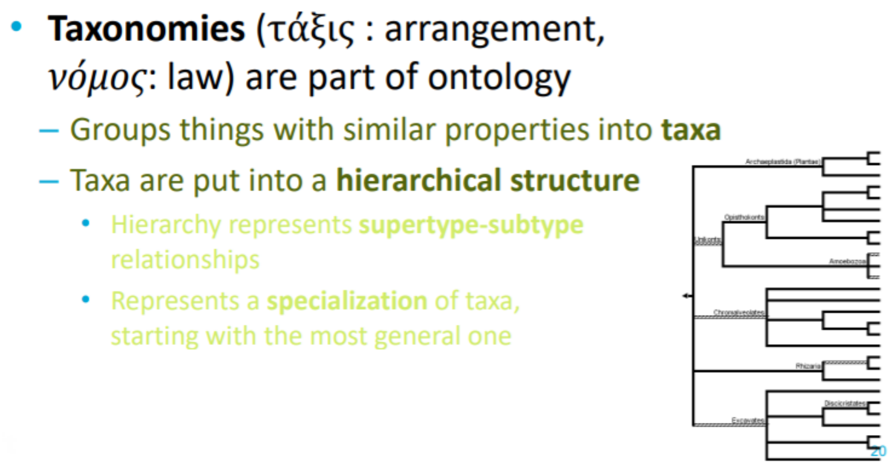
B - **Functional Analysis**
  - concentrates on describing **high-level** user operations and transactions
    - does not yet contain implementation details

B,C,D are software engineering



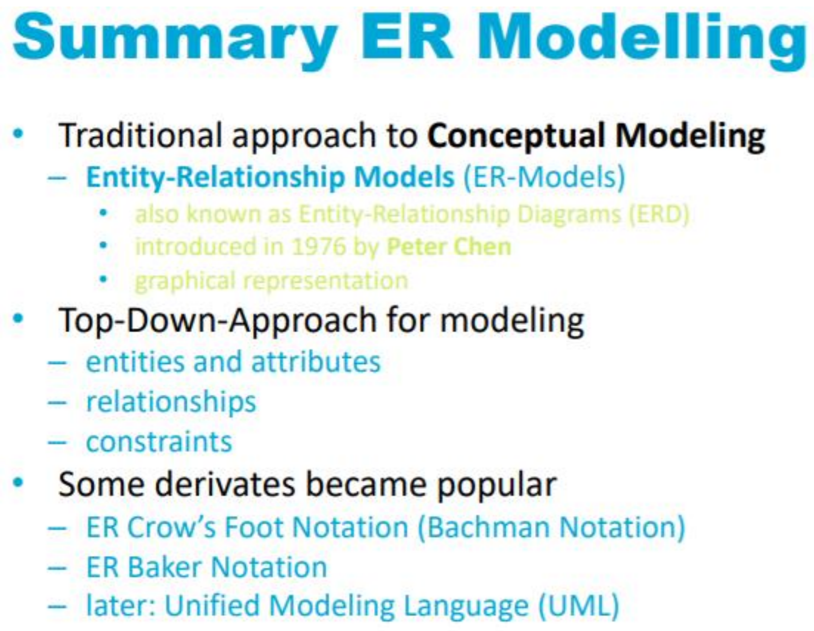
Ontologies

- In computer science ontologies are formal specifications of a shared conceptualization
  - Basically an ontology provides a shared vocabulary and descriptions of the real world
    - It can be used to define the type of objects and/or concepts that exist, and their properties and relations
    - More "expressive" then schemas
    - Ontologies are often equated with taxonomic hierarchies
      - But this definition is far too narrow – ontologies are so much more…
      - See ontologies in philosophy.
  - Domain ontologies model the real world with respect to a specific domain
    - This also disambiguates most terms
    - But domain ontologies are not compatible with each other…



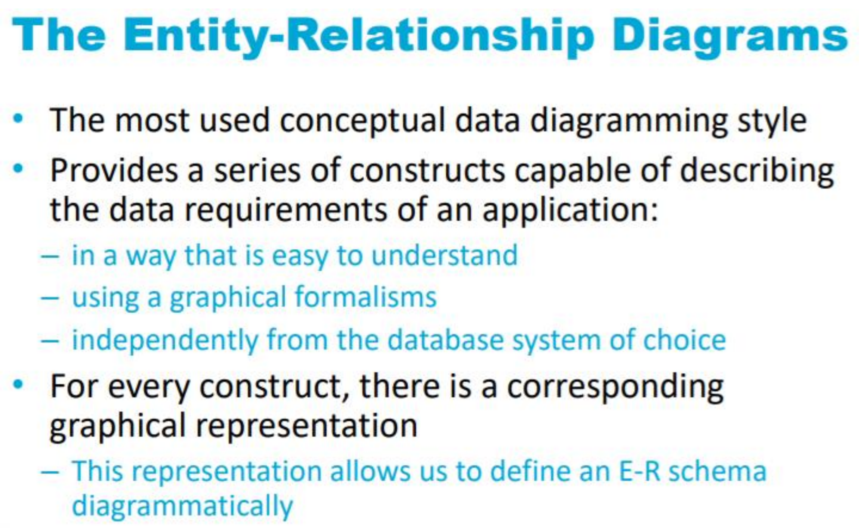
- **Taxonomies** (τάξις : arrangement, *νόμος*: law) are part of ontology
  - Groups things with similar properties into **taxa**
  - Taxa are put into a **hierarchical structure**
    - Hierarchy represents **supertype-subtype** relationships
    - Represents a **specialization** of taxa, starting with the most general one

Archaeplastida (Plantae)
Opisthokonts
Unikonts
Amoebozoa
Chromalveolates
Rhizaria
Discicristates
Excavates
20



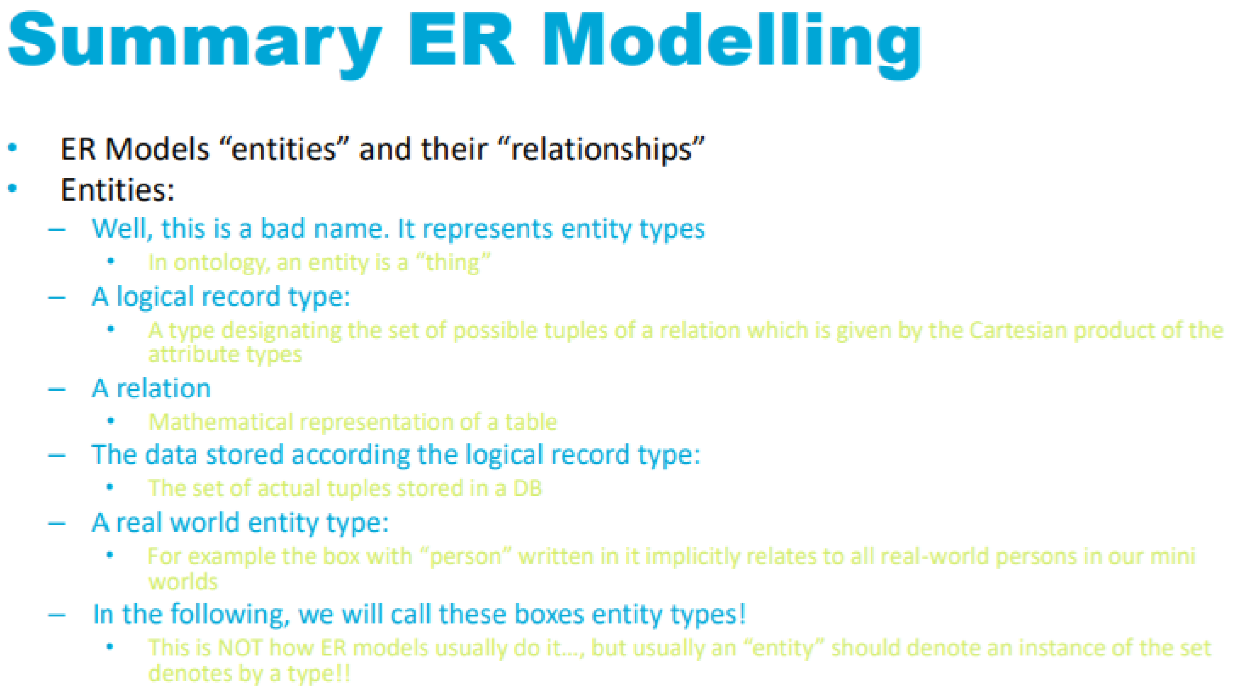
Summary ER Modelling

- Traditional approach to **Conceptual Modeling**
  - **Entity-Relationship Models** (ER-Models)
    - also known as Entity-Relationship Diagrams (ERD)
    - introduced in 1976 by **Peter Chen**
    - graphical representation
- Top-Down-Approach for modeling
  - entities and attributes
  - relationships
  - constraints
- Some derivates became popular
  - ER Crow's Foot Notation (Bachman Notation)
  - ER Baker Notation
  - later: Unified Modeling Language (UML)



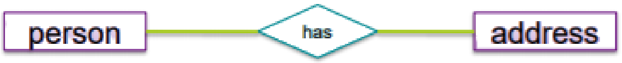
The Entity-Relationship Diagrams

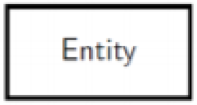
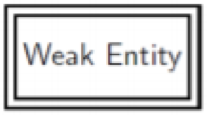
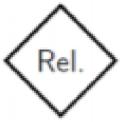
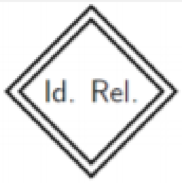
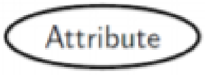
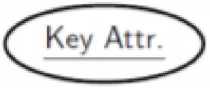
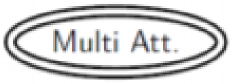
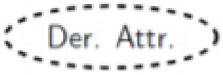
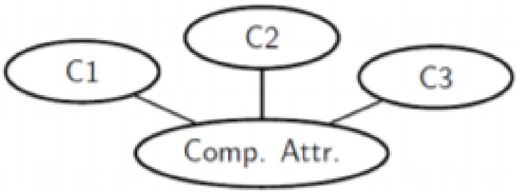
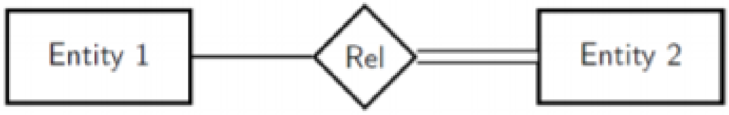
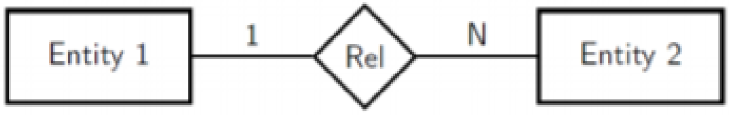
- The most used conceptual data diagramming style
- Provides a series of constructs capable of describing the data requirements of an application:
  - in a way that is easy to understand
  - using a graphical formalisms
  - independently from the database system of choice
- For every construct, there is a corresponding graphical representation
  - This representation allows us to define an E-R schema diagrammatically

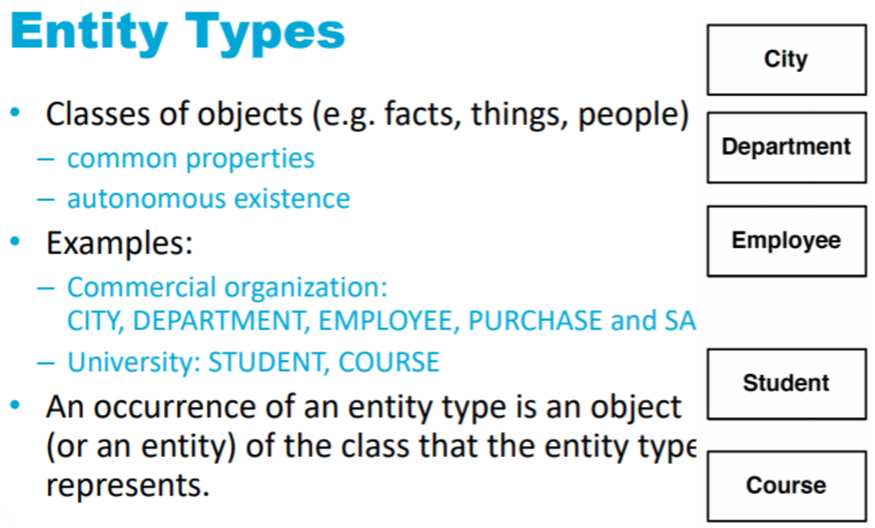


Summary ER Modelling

- ER Models “entities” and their “relationships”
- Entities:
  - Well, this is a bad name. It represents entity types
    - In ontology, an entity is a “thing”
  - A logical record type:
    - A type designating the set of possible tuples of a relation which is given by the Cartesian product of the attribute types
  - A relation
    - Mathematical representation of a table
  - The data stored according the logical record type:
    - The set of actual tuples stored in a DB
  - A real world entity type:
    - For example the box with “person” written in it implicitly relates to all real-world persons in our mini worlds
  - In the following, we will call these boxes entity types!
    - This is NOT how ER models usually do it…, but usually an “entity” should denote an instance of the set denotes by a type!!



Construct	Graphical Representation
Entity	
Weak Entity	
Relationship	
Identifying Relationship	
Attribute	
Key Attribute	
Multi Valued Attribute	
Derived Attribute	
Composite Attribute	
Total Partecipation	
1:N Cardinality	



Entity Types

- Classes of objects (e.g. facts, things, people)
  - common properties
  - autonomous existence
- Examples:
  - Commercial organization: CITY, DEPARTMENT, EMPLOYEE, PURCHASE and SA
  - University: STUDENT, COURSE
- An occurrence of an entity type is an object (or an entity) of the class that the entity type represents.

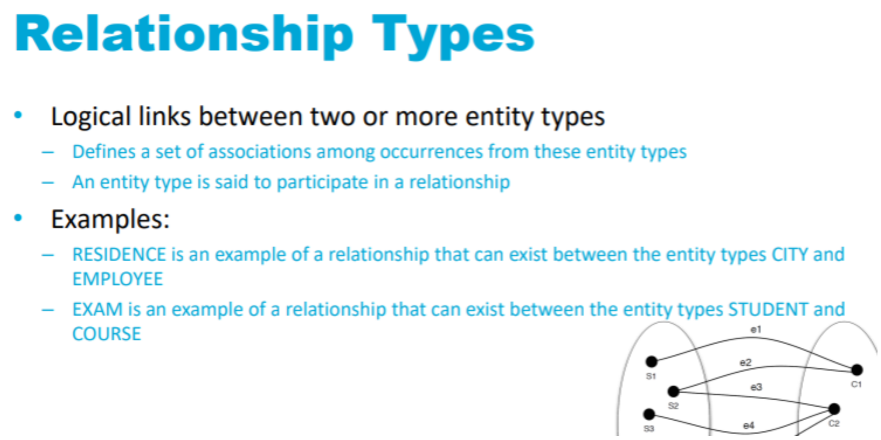
City

Department

Employee

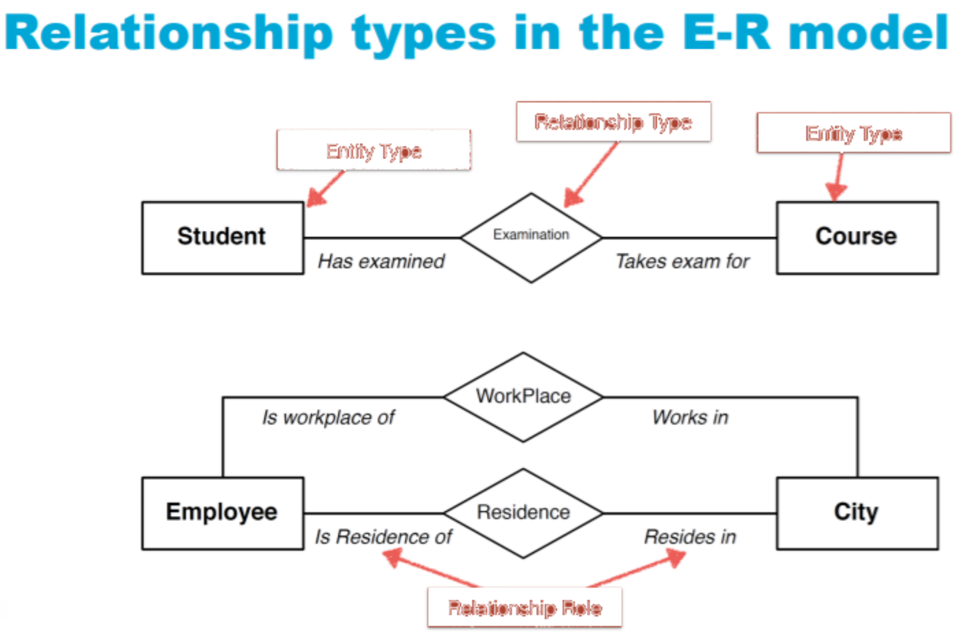
Student

Course



Relationship Types

- Logical links between two or more entity types
  - Defines a set of associations among occurrences from these entity types
  - An entity type is said to participate in a relationship
- Examples:
  - RESIDENCE is an example of a relationship that can exist between the entity types CITY and EMPLOYEE
  - EXAM is an example of a relationship that can exist between the entity types STUDENT and COURSE



Relationship types in the E-R model

Entity Type

Relationship Type

Entity Type

Student

Examination

Course

Has examined

Takes exam for

WorkPlace

Is workplace of

Works in

Employee

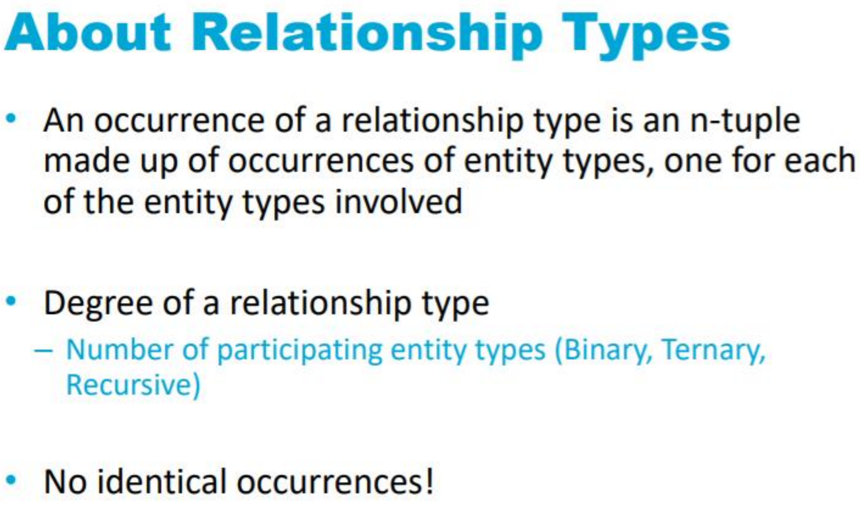
Residence

City

Is Residence of

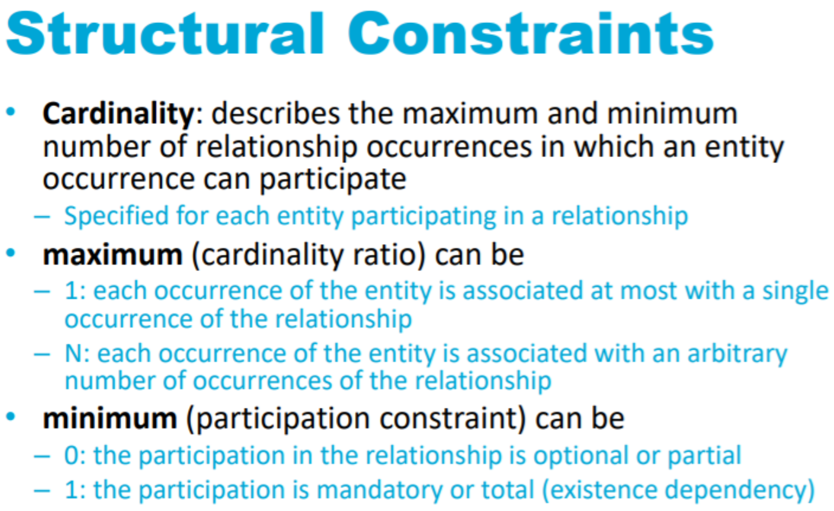
Resides in

Relationship Role



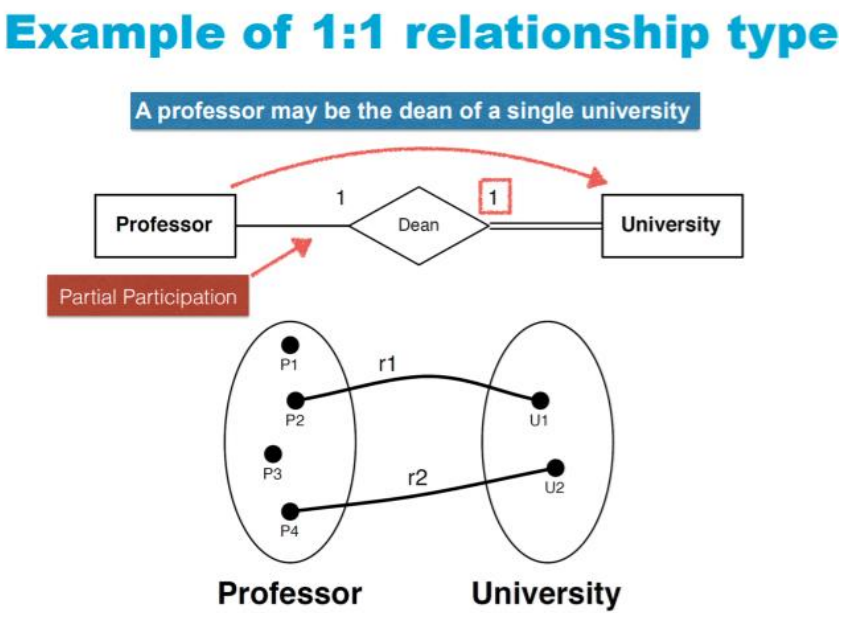
About Relationship Types

- An occurrence of a relationship type is an n-tuple made up of occurrences of entity types, one for each of the entity types involved
- Degree of a relationship type
  - Number of participating entity types (Binary, Ternary, Recursive)
- No identical occurrences!



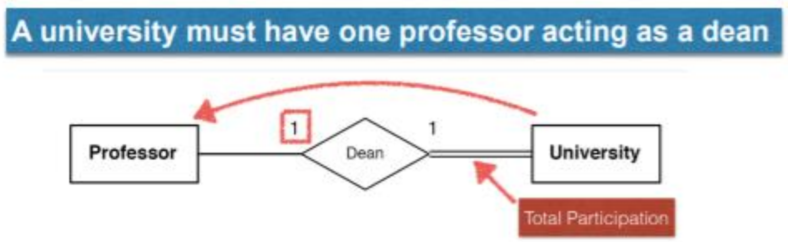
Structural Constraints

- **Cardinality**: describes the maximum and minimum number of relationship occurrences in which an entity occurrence can participate
  - Specified for each entity participating in a relationship
- **maximum** (cardinality ratio) can be
  - 1: each occurrence of the entity is associated at most with a single occurrence of the relationship
  - N: each occurrence of the entity is associated with an arbitrary number of occurrences of the relationship
- **minimum** (participation constraint) can be
  - 0: the participation in the relationship is optional or partial
  - 1: the participation is mandatory or total (existence dependency)



Example of 1:1 relationship type

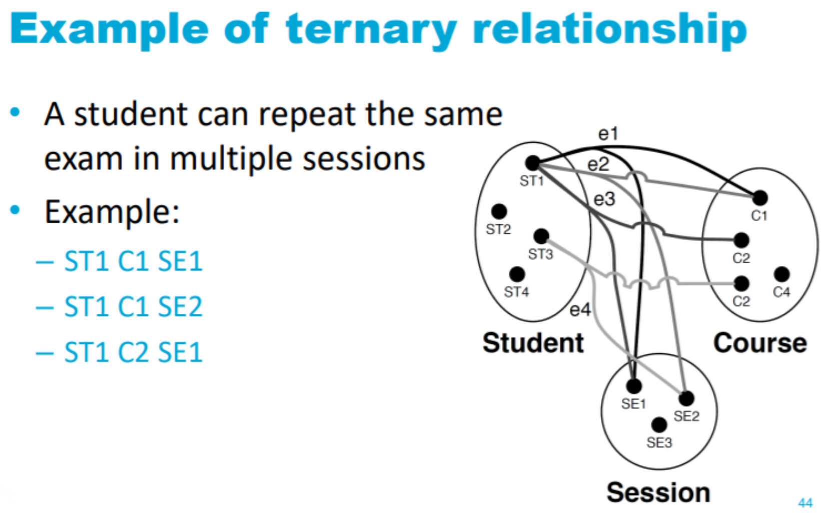
A professor may be the dean of a single university



A university must have one professor acting as a dean

Double line: (must be mapped) All the entities in the university set are mapped to at least 1 professor. All University records must have a Professor Foreign Key (this case 1, but cardinality couuld also be N)

Single line: (optional). Professor records have the Dean_at attribute but can be null

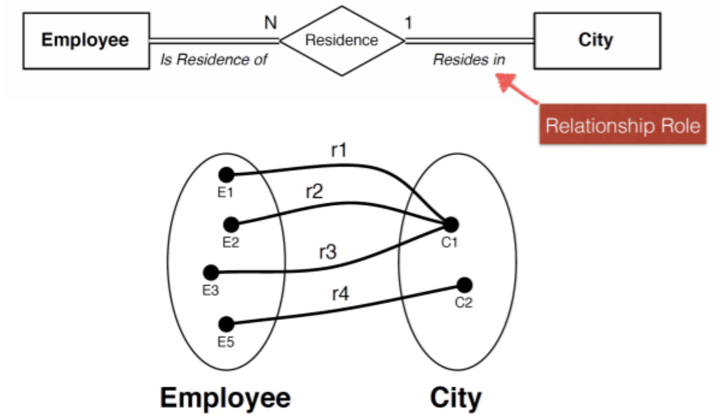


Example of ternary relationship

- A student can repeat the same exam in multiple sessions
- Example:
  - ST1 C1 SE1
  - ST1 C1 SE2
  - ST1 C2 SE1

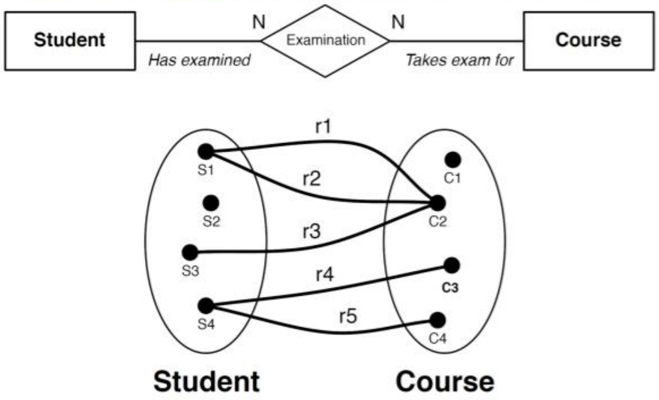
Example of 1:N relationship type

An employee must reside in a city. A city must be the residence of at least one employee



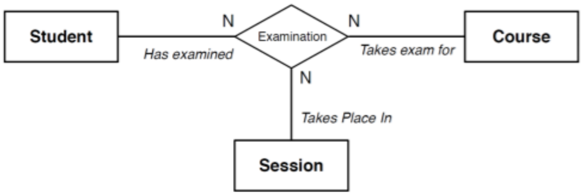
Example of N:N relationship type

A student can take exams for many courses. A course might examine several students.



Student takes the same exam in multiple sessions:

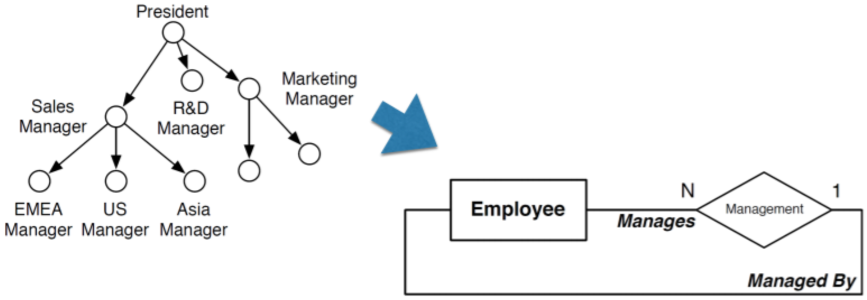
Example of ternary relationship type



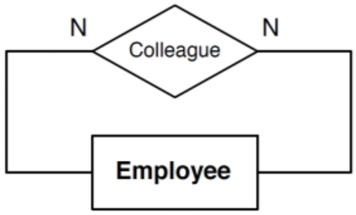
- Maximum cardinalities in an n-ary relationships are almost always N
- If an entity E participates with maximum cardinality 1, it is possible to remove the n-ary relationship and relate E with one of the other entities with a binary relationship

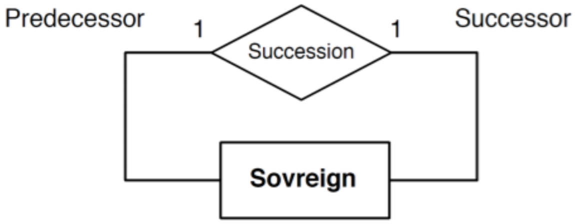
Recursive Relationship Types

- Relationship where the same entity participates with different roles

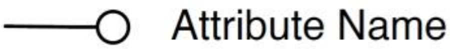


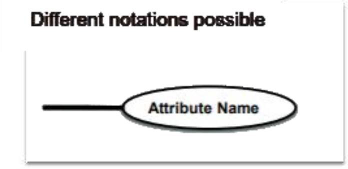
Recursive Relationships Types





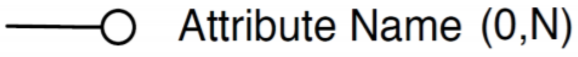
Simple Attributes



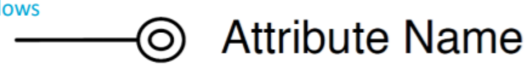


- Describe the elementary properties of entities or relationships
  - Surname, Salary and Age are possible attributes of the EMPLOYEE entity
  - Score is possible attributes for the relationship EXAMINATION between STUDENT and COURSE
- Each attribute is characterized by a domain (not visually represented)

Attribute Cardinality

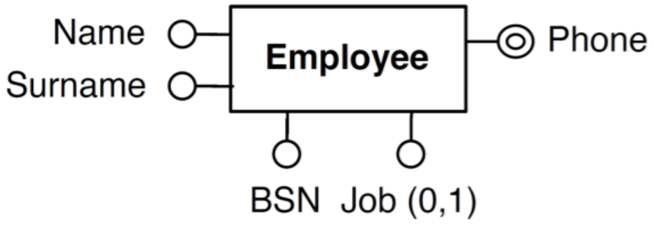


- Describes the minimum and maximum number of values of the attribute associated with each occurrence of an entity or a relationship
  - Typically, cardinality is equal to (1,1) and is omitted
- Minimum cardinality 0 means that the value can be NULL
- Maximum cardinality N means that the attribute may assume more than one value in the same instance
  - In this case we talk about multi-valued attributes and they can be represented as follows

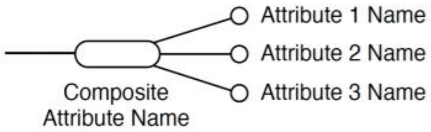


Example

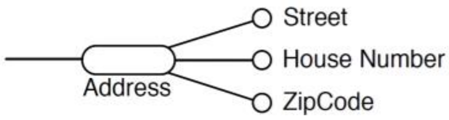
- Beware of multivalued attributes!
  - They might represent situations that could be modelled with additional entity and relationship types
  - E.g. Job as multivalued attribute vs. entity



Composite Attributes

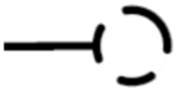


- Achieved by concatenating simpler attribute types
- Pictured by trees of atomic attributes

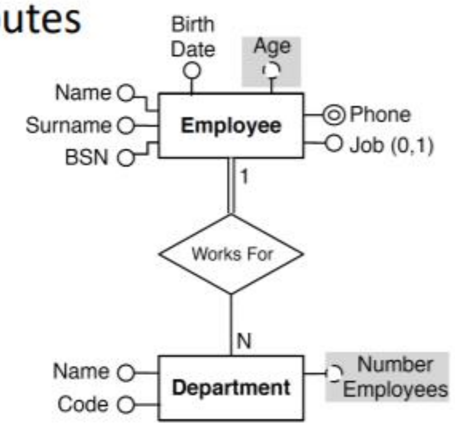


- Composite attributes can form hierarchies

Derived Attributes



- Attributes having values are generated from other attributes
  - calculations, algorithms or procedures
- Examples
  - Age from BirthDate
  - Number of Employee in a department



Attributes of Relationship Types

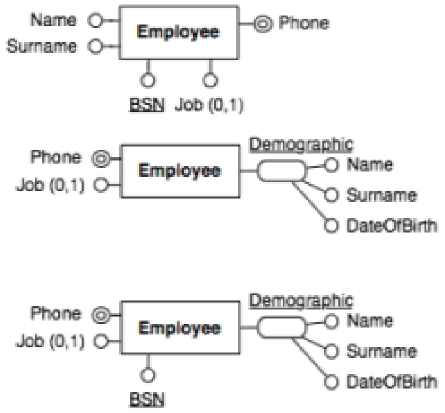
- Attributes of 1:1 relationship types can be migrated to one entity type
- Attributes of 1:N relationship type can be migrated only to entity type on N-side of relationship
- In N:N relationship types, some attributes may be determined by combination of participating entities
  - Must be specified as relationship attributes
  - E.g. Grade for STUDENT and EXAMINATION

Identifiers

- Describe the concepts of the schema that allow the unambiguous identification of the entity instances
  - Attributes and/or entities
- They are specified for each entity of a model
- Relationships have no identifier!

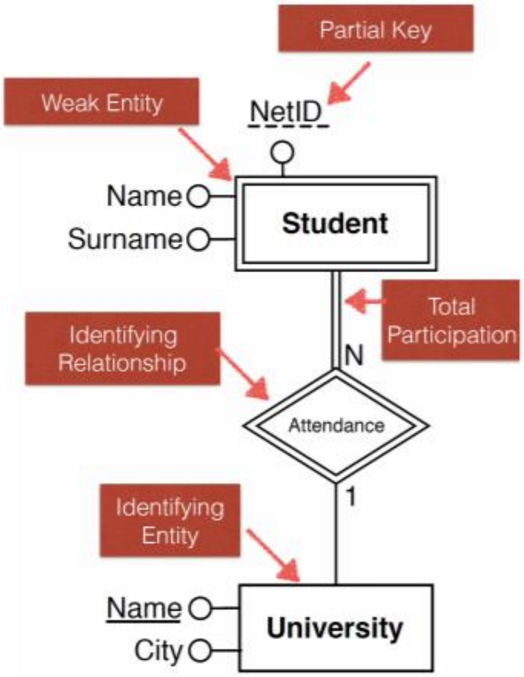
Internal Identifiers

- An identifier is often formed by one or more attributes of the entity itself
- In this case we talk about an internal identifier
  - Also known as a key
- An identifier can involve one or more attributes
  - Each of them has (1,1) cardinality
  - A new composite attribute becomes key
- Each entity must have one identifier, but can have
  more than one
  - Then each distinct underlined attribute is a key
  - There is no primary key in ER (only keys)



Weak Entities

- Sometimes the attributes of an entity are not sufficient to identify its occurrences unambiguously
  - Weak Entities
  - Entities may not have internal identifiers
  - Or partial identifiers (Partial keys)
- Other entities need to be involved in the identification
  - This is called an external identifier
- The relationship that relates a weak entity to its owner is called identifying relationship



Enhanced Entity-Relationship Diagrams (EER)

- Created to design more accurate database schemas reflecting the data properties and constraints more precisely
- More complex requirements than traditional applications
- EER model includes all modelling concepts of the ER model.

- In addition, EER includes:
  - Subclasses and superclasses
  - ~~Specialisation and generalisation~~ (not covered)
  - ~~Category or union type~~ (not covered)
  - ~~Attribute and relationship inheritance~~ (not covered)

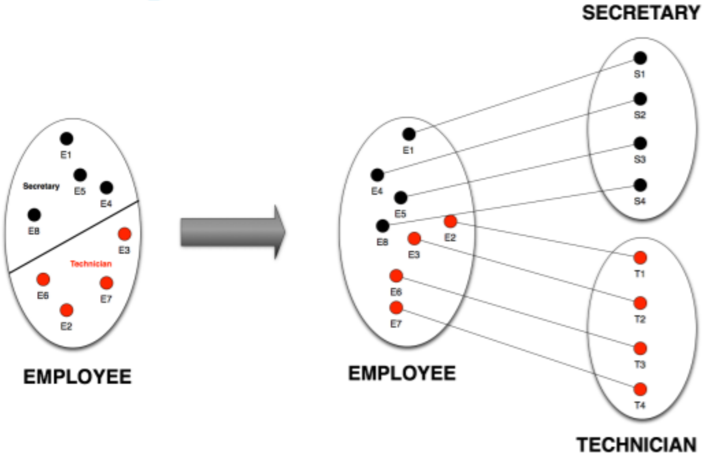
Class / Subclass (Type / Subtype)

- An entity type is used to represent a type of instance (attribute and relationships)
  - EMPLOYEE (name, BSN, BirthDate, etc)

- But also the *collection of entities (entity set) of that type* that exist in the database
  - E.g. entities that represent specific types of employees
  - SECRETARY, TECHNICIANs, MANAGERSs, etc.

- The relationship that exists between a type of entity and its entity set is called a **class/ subclass** relationship
  - EMPLOYEE is the **super type** or **super class**
  - SECRETARY, TECHNICIAN etc. are **subtypes** or **subclasses** of EMPLOYEE

Example Class-Subclass /1

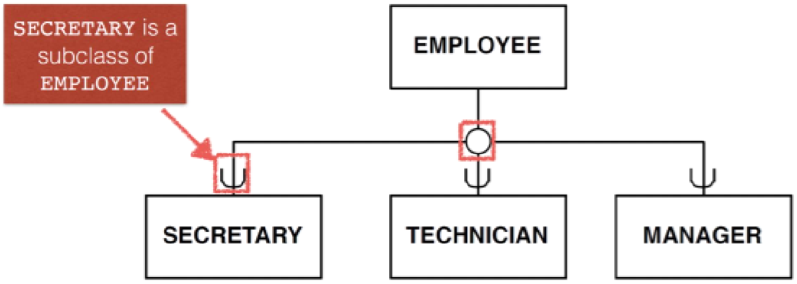


NOT a 1:1 relationship type, but a relationship between OCCURRENCES

Class/ Subclass relationship

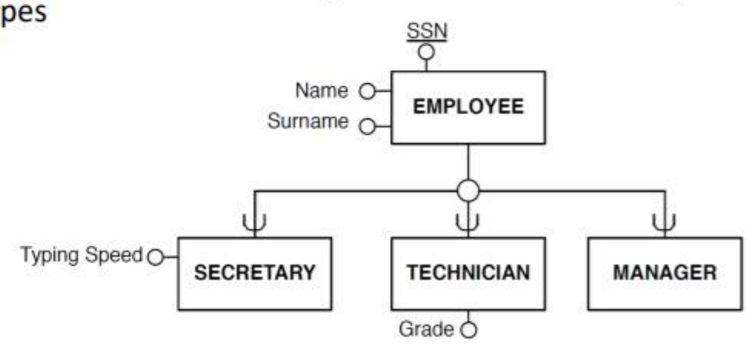
- Also called **generalisation** or **specialisation** relationship
- Represent logical links between an entity E, known as parent entity, and one or more entities $\mathrm{E}_1, \ldots, \mathrm{E}_n$ called child entities, of which E is more general, in the sense that it comprises them as a particular case.
- In this situation we say that E is a **generalisation** of $\mathrm{E}_1, \ldots, \mathrm{E}_n$ and that the entities $\mathrm{E}_1, \ldots, \mathrm{E}_n$ are specialisation of the E entity

- The subset symbol indicates the *direction* of the **specialisation**
- The circle defines the entities involved in the specialisation


SECRETARY is a subclass of EMPLOYEE

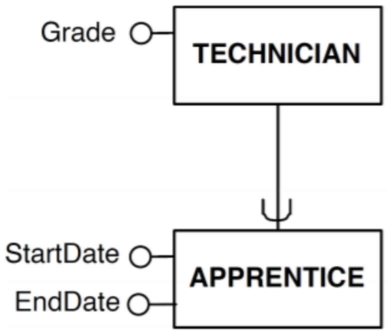
Properties of Specialisation

- Every occurrence of a child entity is also an occurrence of the parent entity
- Inheritance
  - Every property of the parent entity (attributes, identifiers, relationships and other generalisations) is also a property of a child entity.
- Specialisation can define specific attributes and/or specific relationship types



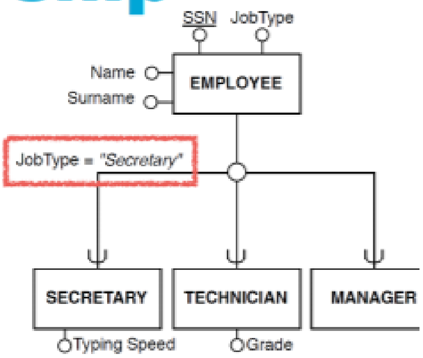
Subset

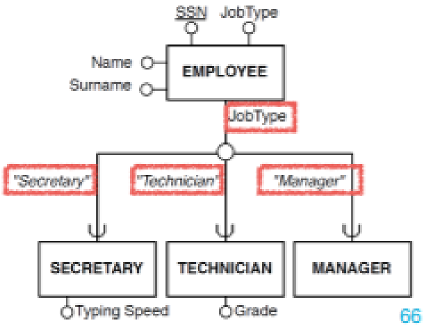
- Special class of specialisation with a single child entity



Constraints: Membership

- In some specialisation we can determine exactly the
  entities that will become members of each subclass
  - Predicate-defined (or condition-defined) subclasses
- All the subclasses in a specialisation have their membership
  condition on the same attribute
  - Attribute-defined specialisation
    (Defining attribute of the specialisation)
- No specific condition
  - User-defined subclass
  - specialisation defined by the database user, instance by instance





Week 2. The Relational Model

Chapter 4. The Extended EER Model

The ER modeling concepts discussed in Chapter 3 are sufficient for representing many database schemas for traditional database applications, which include many data-processing applications in business and industry. However more complex models exist and in this chapter, we describe features that have been proposed for semantic data models and show how the ER model can be enhanced to include these concepts, which leads to the enhanced ER (EER) model.

4.1 Subclasses, Superclasses, and Inheritance

Subtype/subclass entity type

The entities that are members of the EMPLOYEE entity type may be distinguished further into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, and so on. The set or collection of entities in each of the latter groupings is a subset of the entities that belong to the EMPLOYEE entity set, meaning that every entity that is a member of one of these subgroupings is also an employee. We call each of these subgroupings a subclass or subtype of the EMPLOYEE entity type, and the EMPLOYEE entity type is called the superclass or supertype for each of these subclasses.

We call the relationship between a superclass and any one of its subclasses a superclass/subclass or supertype/subtype or simply class/subclass relationship.

The subclass member is the same as the entity in the superclass, but in a distinct specific role

When we implement a superclass/subclass relationship in the database system, however, we may represent a member of the subclass as a distinct database object—say, a distinct record that is related via the key attribute to its superclass entity.

An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass. Such an entity can be included optionally, as a member of any number of subclasses. Because an entity in the subclass represents the same real-world entity from the superclass, it should possess values for its specific attributes as well as values of its attributes as a member of the superclass. We say that an entity that is a member of a subclass inherits all the attributes of the entity as a member of the superclass. The entity also inherits all the relationships in which the superclass participates.

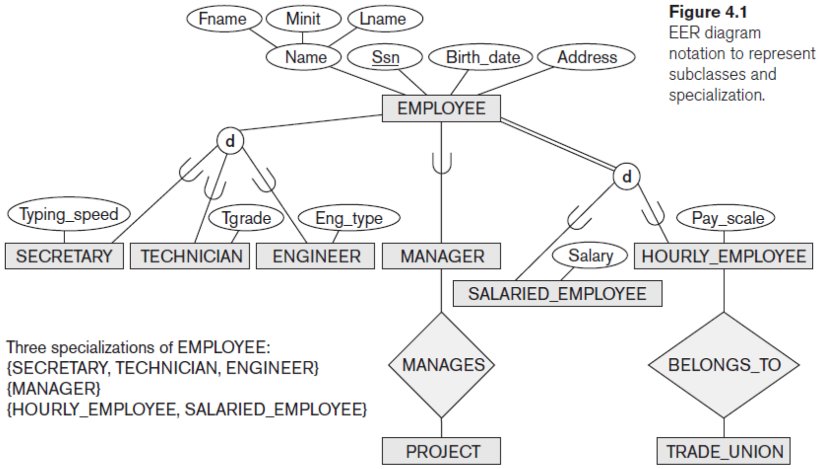


Figure 4.1
EER diagram notation to represent subclasses and specialization.

Three specializations of EMPLOYEE:
{SECRETARY, TECHNICIAN, ENGINEER}
{MANAGER}
{HOURLY_EMPLOYEE, SALARIED_EMPLOYEE}

Observe how the subclass Manager is a subset of super class Employee
Manager ⊂ Employee

And see how the letter d (disjoint/at most) expresses that an employ can be secretary xor technician xor engineer **xor none**; and on top of that it can be or not be a manager.The double line means total participation. So all employees are registered as either salried or hourly, whereas not all employees are registered as secretary, technicitan or engineer, that is, there are some positions in the company that do not have an entity type of their own.

The diamonds are relationships. In Chen notation (this one, the one from the book), relations and cardinalities are represented in active voice. “Many employees have 1 mentor”.

4.2 Specialization and Generalization

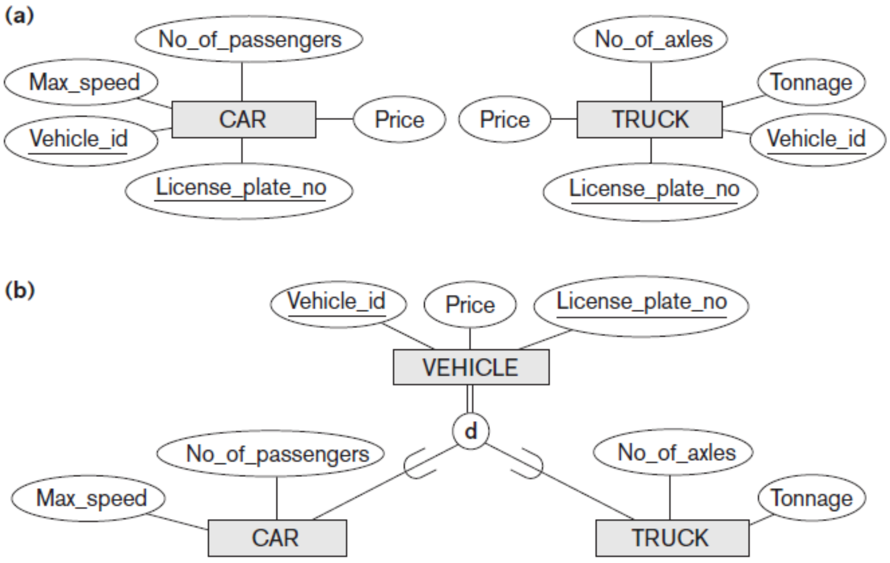
4.2.1 Specialization

Specialization is the process of defining a set of subclasses of an entity type. The set of subclasses that forms a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass. Attributes that apply only to entities of a particular subclass—such as TypingSpeed of SECRETARY—are attached to the rectangle representing that subclass. These are called specific (or local) attributes of the subclass.

4.2.2 Generalization

we suppress the differences among several entity types, identify their common features, and generalize them into a single superclass of which the original entity types are special subclasses.

Figure 4.3
Generalization. (a) Two entity types, CAR and TRUCK.
(b) Generalizing CAR and TRUCK into the superclass VEHICLE.



4.3 Constraints and Characteristics of Specialization and Generalization Hierarchies

4.3.1 Constraints on Specialization and Generalization

In some specializations we can determine exactly the entities that will become members of each subclass by placing a condition on the value of some attribute of the superclass. Such subclasses are called predicate-defined (or condition-defined) subclasses.

For example, if the EMPLOYEE entity type has an attribute Job_type we can specify the condition of membership in the SECRETARY subclass by the condition (Job_type = 'Secretary'), which we call the defining predicate of the subclass. This condition is a constraint.

When we do not have a condition for determining membership in a subclass, the subclass is called user-defined. membership is specified individually for each entity by the user, not by any condition
that may be evaluated automatically.
The d notation also applies to user-defined subclasses of a specialization that must be disjoint.

Subclasses can also be overlapping (o) that is, the same (real-world) entity may be a member of more than one subclass of the specialization.

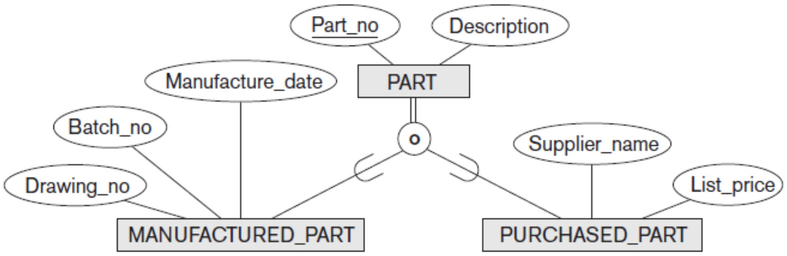


Figure 4.5
EER diagram notation for an overlapping (nondisjoint) specialization.

The second constraint on specialization is called the completeness (or totalness) constraint, which may be total or partial. A total specialization constraint specifies that every entity in the superclass must be a member of at least one subclass in the specialization. For example, if every EMPLOYEE must be either an HOURLY_EMPLOYEE or a SALARIED_EMPLOYEE, then the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Figure 4.1 is a total specialization of EMPLOYEE. This is shown in EER diagrams by using a double line to connect the superclass to the circle. A single line is used to display a partial specialization, which allows an entity not to belong to any of the subclasses.

4.3.2 Specialization and Generalization Hierarchies and Lattices

A specialization **hierarchy** (**single inheritance**) has the constraint that every subclass participates as a subclass in only one class/subclass relationship; that is, each subclass has only one parent, which results in a tree structure or strict hierarchy.

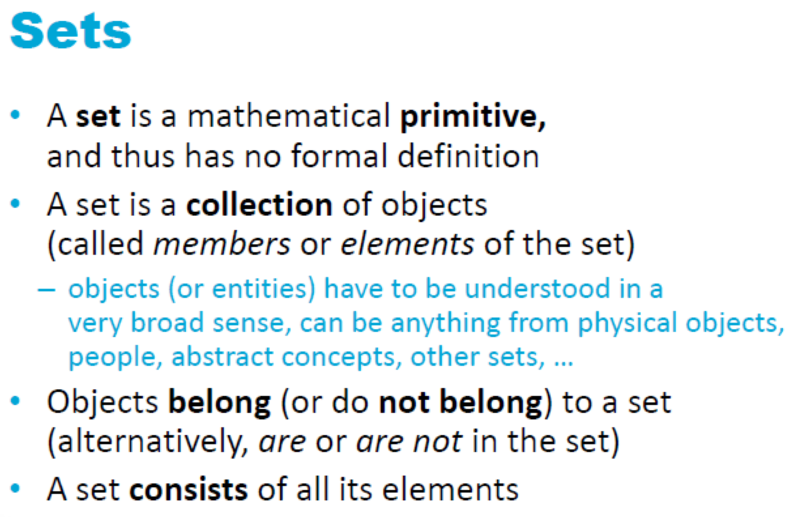
For a specialization **lattice** (**mulitple inheritance**), a subclass can be a subclass in more than one class/subclass relationship. That is a shared subclass. if no shared subclasses existed, we would have a hierarchy rather than a lattice and only single inheritance would exist.

4.7 Ontology

The main difference between an ontology and, say, a database schema, is that the schema is usually limited to describing a small subset of a miniworld from reality in order to store and manage data. An ontology is usually considered to be more general in that it attempts to describe a part of reality or a domain of interest (for example, medical terms, electronic-commerce applications, sports, and so on) as completely as possible.

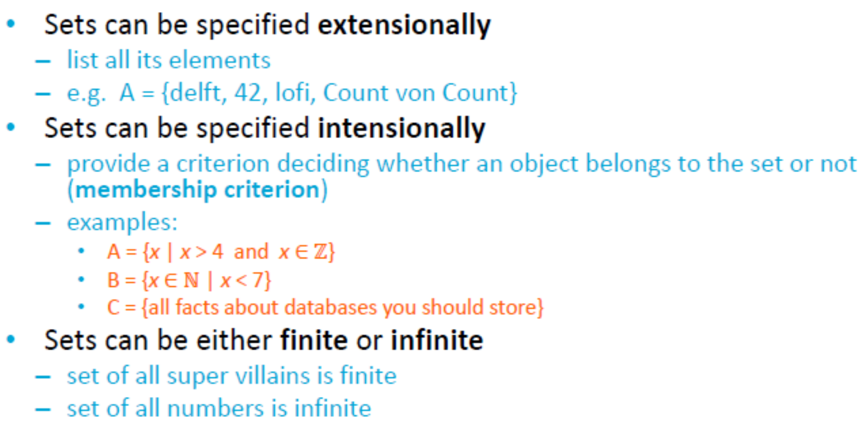
Lecture 3. The Relational Model

The relational model is based on set theory, which is the foundation of mathematics.

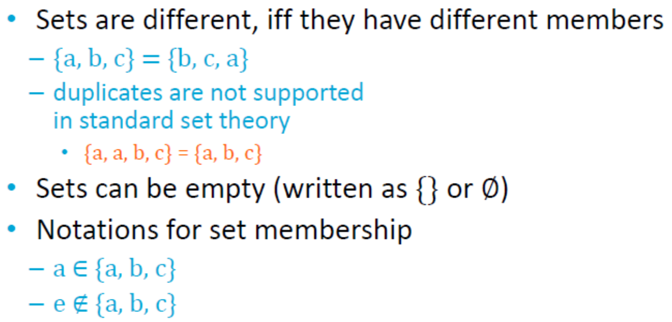


Sets

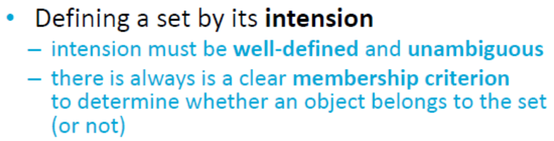
- A **set** is a mathematical **primitive,** and thus has no formal definition
- A set is a **collection** of objects (called *members* or *elements* of the set)
  - objects (or entities) have to be understood in a very broad sense, can be anything from physical objects, people, abstract concepts, other sets, …
- Objects **belong** (or do **not belong**) to a set (alternatively, *are* or *are not* in the set)
- A set **consists** of all its elements



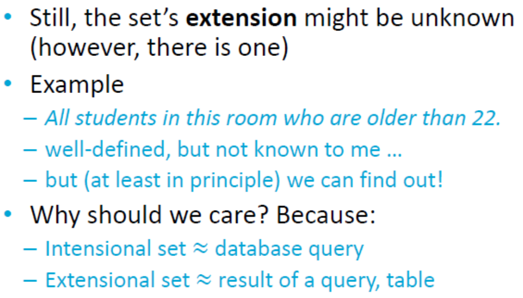
- Sets can be specified **extensionally**
  - list all its elements
  - e.g. A = {delft, 42, lofi, Count von Count}
- Sets can be specified **intensionally**
  - provide a criterion deciding whether an object belongs to the set or not (**membership criterion**)
  - examples:
    - $A = \{x \mid x > 4 \text{ and } x \in \mathbb{Z}\}$
    - $B = \{x \in \mathbb{N} \mid x < 7\}$
    - C = {all facts about databases you should store}
- Sets can be either **finite** or **infinite**
  - set of all super villains is finite
  - set of all numbers is infinite



- Sets are different, iff they have different members
  - $\{a, b, c\} = \{b, c, a\}$
  - duplicates are not supported in standard set theory
    - $\{a, a, b, c\} = \{a, b, c\}$
- Sets can be empty (written as {} or ∅)
- Notations for set membership
  - $a \in \{a, b, c\}$
  - $e \notin \{a, b, c\}$

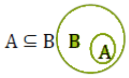


- Defining a set by its **intension**
  - intension must be **well-defined** and **unambiguous**
  - there is always is a clear **membership criterion** to determine whether an object belongs to the set (or not)



- Still, the set's **extension** might be unknown (however, there is one)
- Example
  - *All students in this room who are older than 22.*
  - well-defined, but not known to me …
  - but (at least in principle) we can find out!
- Why should we care? Because:
  - Intensional set ≈ database query
  - Extensional set ≈ result of a query, table

- Sets have a cardinality (i.e., number of elements)
  - denoted by $|A|$
  - $|\{a, b, c\}| = 3$



- Set A is a **subset** of set B, denoted by $A \subseteq B$, iff every member of A is also a member of B
- B is a **superset** of A, denoted by $B \supseteq A$, iff $A \subseteq B$

Tuples

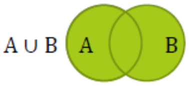
- A **tuple** (or vector) is a sequence of objects
  - length 1: Singleton
  - length 2: Pair
  - length 3: Triple
  - length *n*: *n*-tuple
- In contrast to sets…
  - tuples can contain an object more than once
  - the objects appear in a certain **order**
  - the length of the tuple is **finite**
- Written as $\langle a, b, c\rangle$ or (a, b, c)

- Hence
  - $\langle a, b, c\rangle \neq \langle c, b, a\rangle$, whereas $\{a, b, c\} = \{c, b, a\}$
  - $\langle a_1, a_2\rangle = \langle b_1, b_2\rangle$ iff $a_1 = b_1$ and $a_2 = b_2$

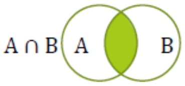
- ***n*-tuples** ($n > 1$) can also be defined as a cascade of ordered pairs:
  - $\langle a, b, c, d\rangle = \langle a, \langle b, \langle c, d\rangle\rangle\rangle$

Set Operations

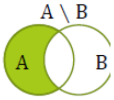
- Four binary **set operations**
  - union, intersection, difference and cartesian product
- Union: ∪
  - creates a new set containing all elements that are contained in (at least) one of two sets
  - $\{a, b\} \cup \{b, c\} = \{a, b, c\}$



- Intersection: ∩
  - creates a new set containing all elements that are contained in both sets
  - $\{a, b\} \cap \{b, c\} = \{b\}$



- Difference: \
  - creates a set containing all elements of the first set without those also being in the second set
  - $\{a, b\} \setminus \{b, c\} = \{a\}$



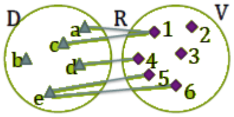
- Cartesian Product: ×
  - the cartesian product is an operation between two sets, creating a new set of pairs such that:
    $$A \times B = \{\langle a, b\rangle \mid a \in A \text{ and } b \in B\}$$
  - named after René Descartes
- Example
  - $\{a, b\} \times \{b, c\} = \{\langle a, b\rangle, \langle a, c\rangle, \langle b, b\rangle, \langle b, c\rangle\}$

Relations

- A **relation** R over some sets $D_1, \ldots, D_n$ is a **subset** of their **cartesian** product
  - $R \subseteq D_1 \times \ldots \times D_n$
  - the elements of a relation are **tuples**
  - the D_i are called **domains**
  - each D_i corresponds to an **attribute** of a tuple
    - *n*=1: Unary relation or **property**
    - *n*=2: Binary relation
    - *n*=3: Ternary relation
    - …

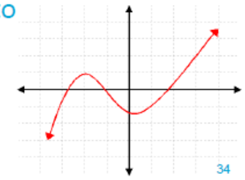
- Some important properties
  - relations are sets (of tuples) in the mathematical sense, thus **no duplicate tuples** are allowed
  - the **set of tuples** is **unordered**
  - the **list of domains** is **ordered**
    - And so are the values of each tuple
  - relations can be modified by…
    - **inserting** new tuples,
    - **deleting** existing tuples, and
    - **updating** (that is, modifying) existing tuples.

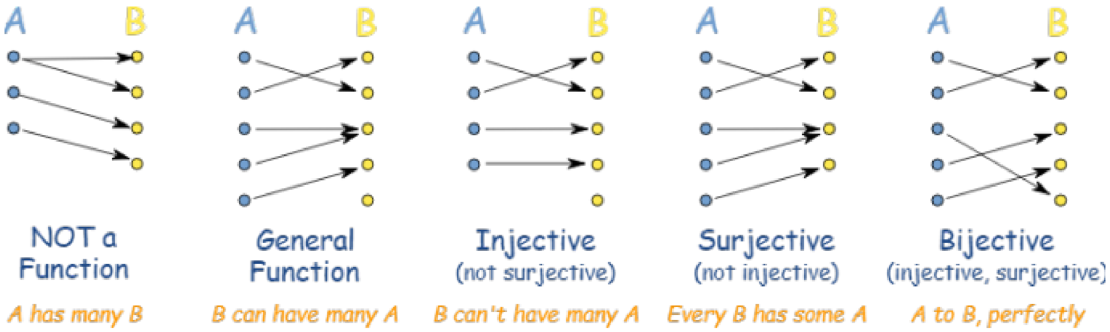
- A special case: Binary relations
  - $R \subseteq D_1 \times D_2$
    - D_1 is called **domain,** D_2 is called **co-domain** (range, target)
  - relates objects of two different sets to each other
  - R is just a set of ordered pairs
  - $R = \{\langle a,1\rangle, \langle c,1\rangle, \langle d,4\rangle, \langle e,5\rangle, \langle e,6\rangle\}$
    - can also be written as a**R**1, c**R**1, d**R**4, …
  - imagine Likes ⊆ Person × Beverage
    - Asterios Likes Coffee, Christoph Likes Tea, …
  - For example, binary relations can naively be used to implement n:m relationship types in a logical data model
  - Functions are a special case of binary relations



Functions

- **Functions** are special case of binary relations
  - **partial function:**
    each element of the domain is related to
    at most one element in the co-domain
  - **total function:**
    each element in the domain is related to
    exactly one element in the co-domain





Partial function = injective

Total function = bijective

- Functions can be used to **abstract** from the exact order of domains in a relation
  - alternative definition of relations:
    a relation is a set of functions
  - every tuple in the relation is considered as a function of the type $\{A_1, \ldots, A_n\} \rightarrow D_1 \cup \ldots \cup D_n$
    - that means, every tuple maps each attribute to some value

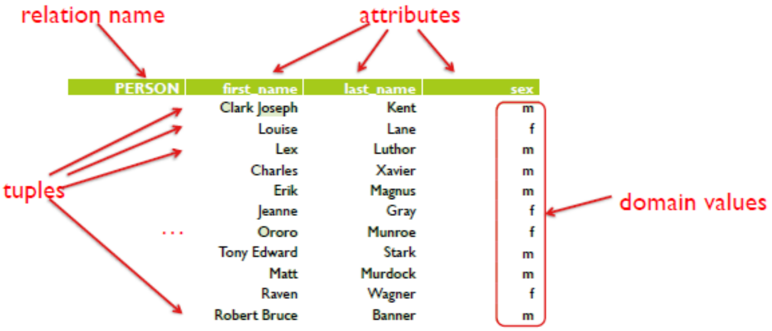
- Example
  - Color = {pink, black}
  - Material = {silk, armor plates}
  - Accessory = {spikes, butterfly helmet}
  - to be independent of the domain order, the tuple ⟨pink, silk, butterfly helmet⟩ can also be represented as the following function *t*
    - *t*(Color) = pink
    - *t*(Material) = silk
    - *t*(Accessory) = butterfly helmet
  - Usually, one writes t[color] instead of t(color)
  - This can be used to change the order of domains for tuples
    - t[Material, Accessory, Color] = ⟨silk, butterfly helmet, pink⟩

- a **database schema** is a description in terms of relations and attribute domains
  - Description of all possible instances
  - ...but also intensional set of tuples
- a **database instance** is a set of tuples having certain attribute values
  - An extensional set of tuples

Relational Model

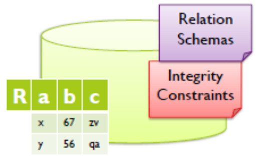
- Database schemas are described by **relation schemas** $\mathrm{R}(\mathrm{A}_1, \ldots, \mathrm{A}_n)$
- Domains are assigned by the dom function
  - $\mathrm{dom}(\mathrm{A}_1) = \mathrm{D}_1, \mathrm{dom}(\mathrm{A}_2) = \mathrm{D}_2, \ldots$
  - Also written as: $\mathrm{R}(\mathrm{A}_1{:}\mathrm{D}_1, \ldots, \mathrm{A}_n{:}\mathrm{D}_n)$
- The actual database instance is given by a set of matching **relations**
- Example
  - relation schema: Cat(name: string, age: number)
  - A matching relation: { (Blackie, 2), (Kitty, 1), (Fluffy, 4) }

Relational Model



PERSON	first_name	last_name	sex
	Clark Joseph	Kent	m
	Louise	Lane	f
	Lex	Luthor	m
	Charles	Xavier	m
	Erik	Magnus	m
	Jeanne	Gray	f
...	Ororo	Munroe	f
	Tony Edward	Stark	m
	Matt	Murdock	m
	Raven	Wagner	f
	Robert Bruce	Banner	m

- A **relational database schema** consists of
  - a set of relation schemas
  - a set of integrity constraints
- A **relational database instance** (or state) is
  - a set of relations adhering to the respective schemas and respecting all integrity constraints



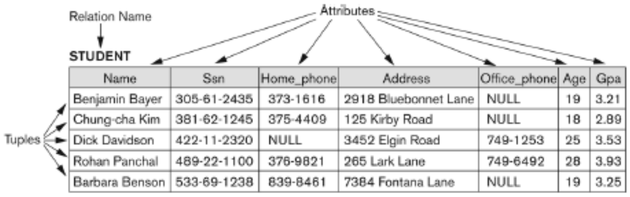
- Every relational DBMS needs a language to define its relation schemas (and integrity constraints)
  - **Data Definition Language** (DDL)
  - typically, it is difficult to formalize all possible integrity constraints, since they tend to be complex and vague
- A relational DBMS also needs a language to handle and manipulate tuples
  - **Data Manipulation Language** (DML)
- Today's RDBMS use **SQL** as both DDL and DML
  - Compare to XML: Here, DDL and DML are separated

Relational Model Concepts

- The relational model represents the database as a collection of relations
- Each relation resembles a table of values
- When a relation is thought of as a table of values, each row in the table represents a collection of related data values

Formal Terminology

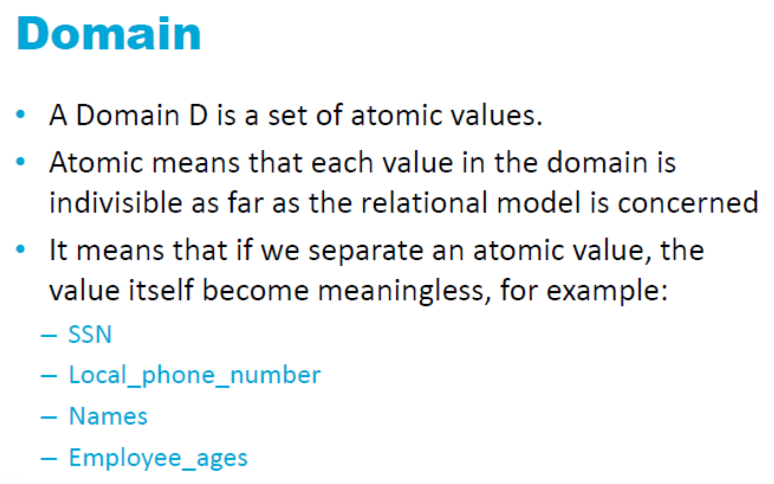
- A row is called a tuple
- A column header is called an attribute
- The table is called relation



STUDENT

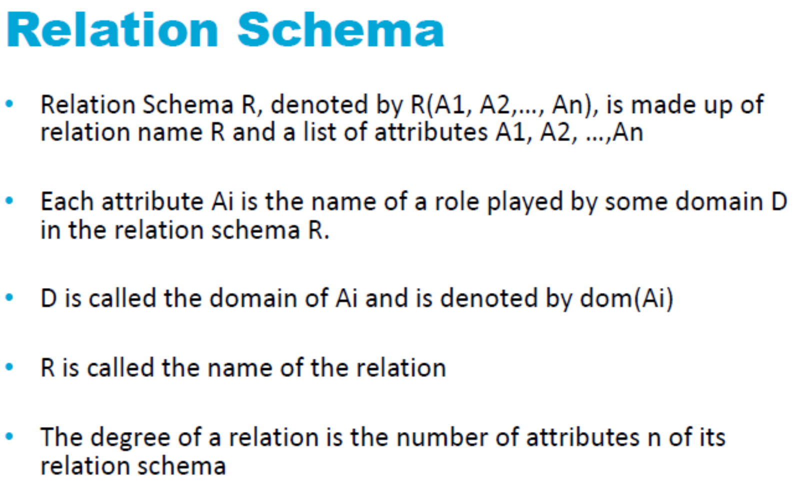
Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	749-1253	25	3.53
Rohan Panchal	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	NULL	19	3.25

Figure 5.1
The attributes and tuples of a relation STUDENT.



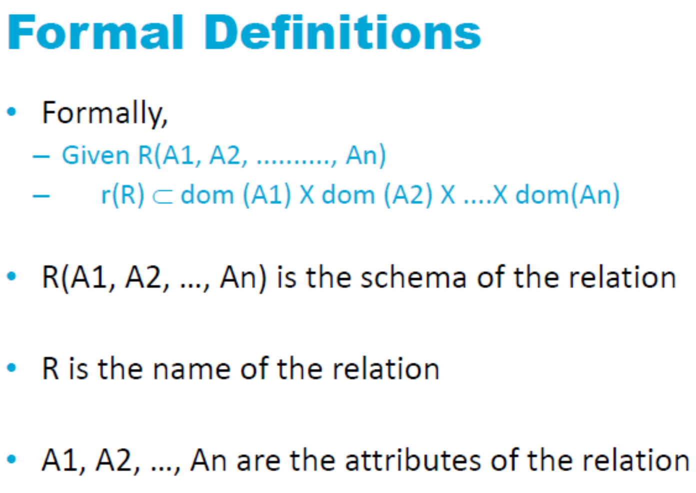
Domain

- A Domain D is a set of atomic values.
- Atomic means that each value in the domain is indivisible as far as the relational model is concerned
- It means that if we separate an atomic value, the value itself become meaningless, for example:
  - SSN
  - Local_phone_number
  - Names
  - Employee_ages



Relation Schema

- Relation Schema R, denoted by R(A1, A2,..., An), is made up of relation name R and a list of attributes A1, A2, ...,An
- Each attribute Ai is the name of a role played by some domain D in the relation schema R.
- D is called the domain of Ai and is denoted by dom(Ai)
- R is called the name of the relation
- The degree of a relation is the number of attributes n of its relation schema



Formal Definitions

- Formally,
  - Given R(A1, A2, ..., An)
  - r(R) ⊂ dom (A1) X dom (A2) X ....X dom(An)
- R(A1, A2, ..., An) is the schema of the relation
- R is the name of the relation
- A1, A2, ..., An are the attributes of the relation

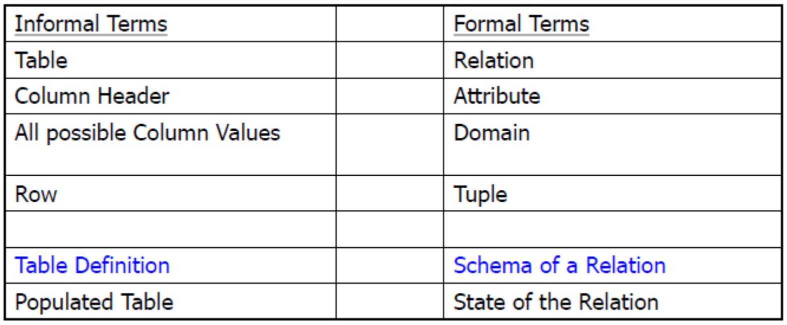
Formal Definitions

- Let R(A1, A2) be a relation schema:
  - Let dom(A1) = {0,1}
  - Let dom(A2) = {a,b,c}

- Then: dom(A1) X dom(A2) is all possible combinations:
  - {<0,a> , <0,b> , <0,c>, <1,a>, <1,b>, <1,c> }

- The relation state r(R) ⊂ dom(A1) X dom(A2)

Definition Summary

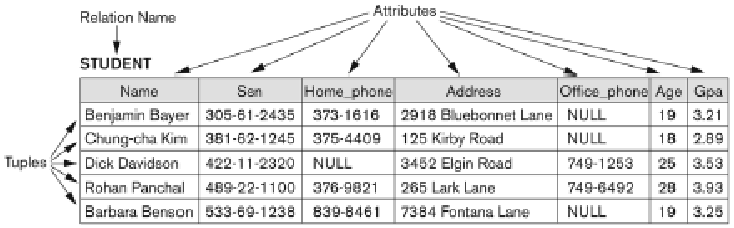


Informal Terms		Formal Terms
Table		Relation
Column Header		Attribute
All possible Column Values		Domain
Row		Tuple
Table Definition		Schema of a Relation
Populated Table		State of the Relation

Ordering

- **Ordering of Tuples is a Relation**
  - a relation is defined as a set of tuples.
  - Mathematically, elements of a set have **NO** order among them
  - The ordering indicates first, second, ith, and last records in the file
  - Hence, the following two relations are **identical**

Identical Relations

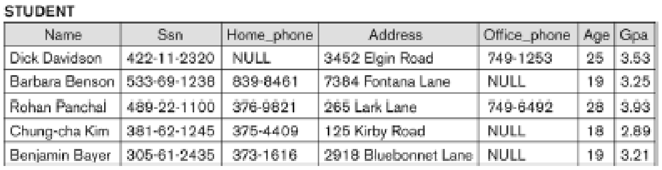

Relation Name
Attributes
Tuples

STUDENT

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	749-1253	25	3.53
Rohan Panchal	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	NULL	19	3.25

Figure 5.1
The attributes and tuples of a relation STUDENT.

Figure 5.2
The relation STUDENT from Figure 5.1 with a different order of tuples.



STUDENT

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	749-1253	25	3.53
Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	NULL	19	3.25
Rohan Panchal	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
Chung-cha Kim	381-62-1245	375-4409	125 Kirby Road	NULL	18	2.89
Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	NULL	19	3.21

Values in the Tuples

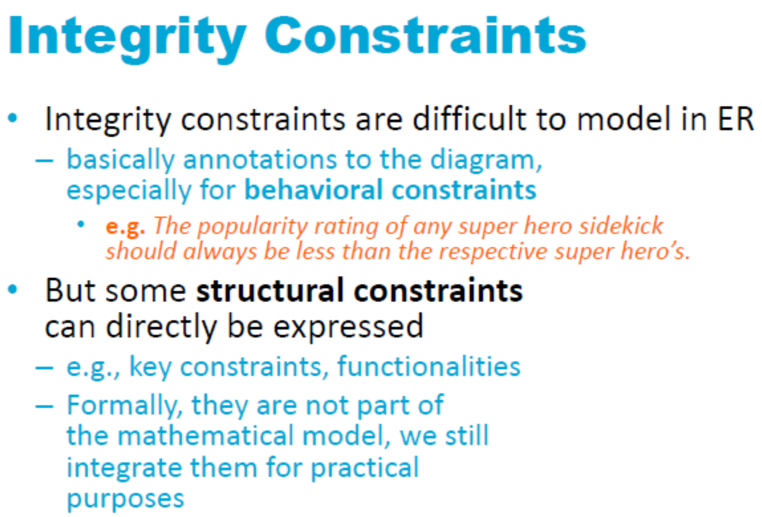
- Each value in a tuple is an atomic value
- Hence, composite and multi-valued attributes are not allowed
- This model is sometimes called the flat relational model
- Much of the theory behind the relational model was developed with this assumption, which is called first normal form assumption

Null in tuples

- An important concept is that if NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple

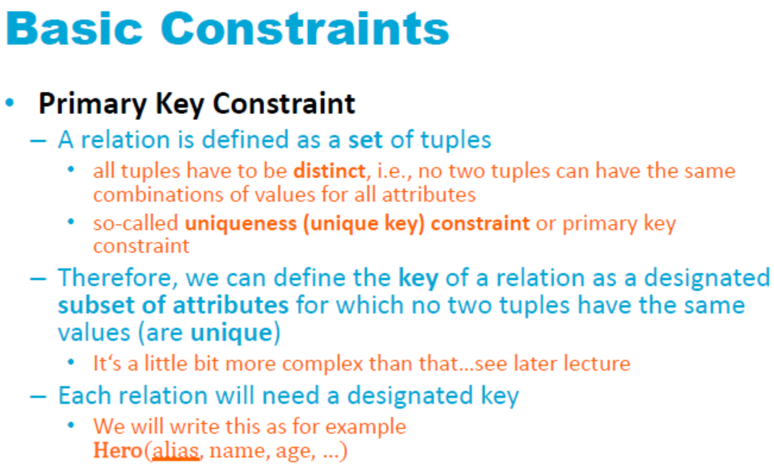
Relational Model Notation

- An attribute A can be qualified with the relation name R to which it belongs by using the dot notation R.A
- For example, STUDENT.Name or STUDENT.Age



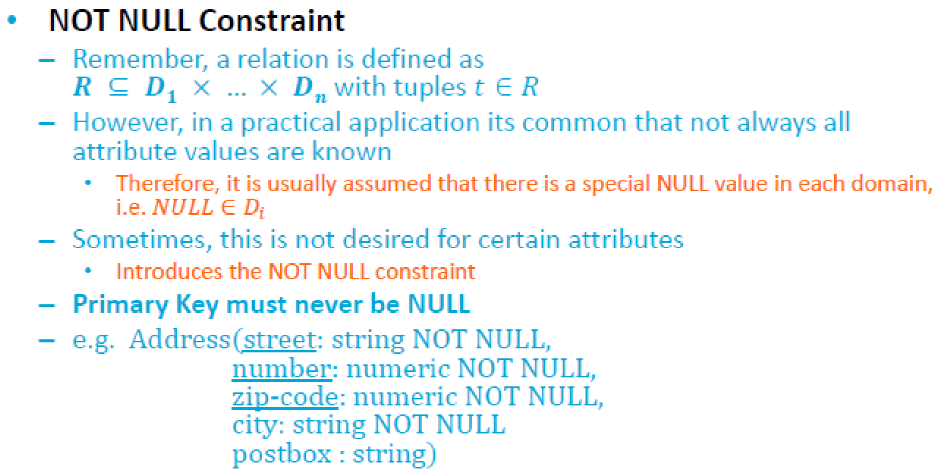
Integrity Constraints

- Integrity constraints are difficult to model in ER
  - basically annotations to the diagram, especially for **behavioral constraints**
    - **e.g.** *The popularity rating of any super hero sidekick should always be less than the respective super hero's.*
- But some **structural constraints** can directly be expressed
  - e.g., key constraints, functionalities
  - Formally, they are not part of the mathematical model, we still integrate them for practical purposes

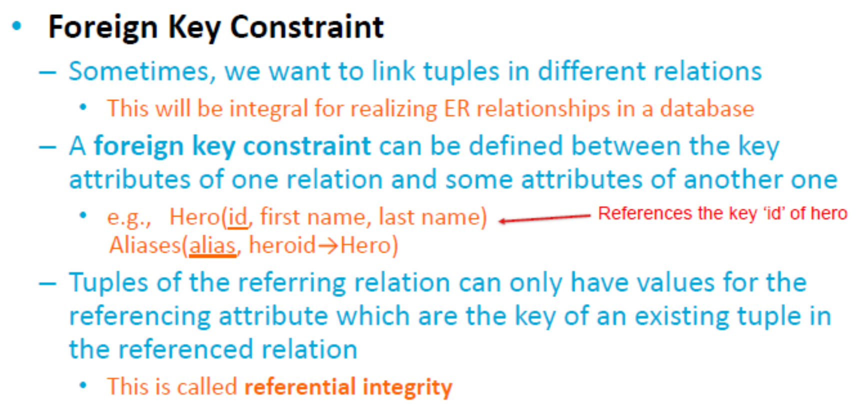


Basic Constraints

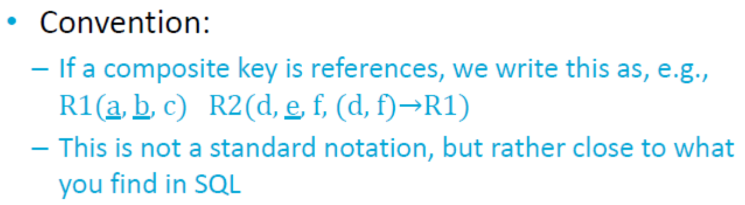
- **Primary Key Constraint**
  - A relation is defined as a **set** of tuples
    - all tuples have to be **distinct**, i.e., no two tuples can have the same combinations of values for all attributes
    - so-called **uniqueness (unique key) constraint** or primary key constraint
  - Therefore, we can define the **key** of a relation as a designated **subset of attributes** for which no two tuples have the same values (are **unique**)
    - It's a little bit more complex than that…see later lecture
  - Each relation will need a designated key
    - We will write this as for example **Hero**(alias, name, age, …)



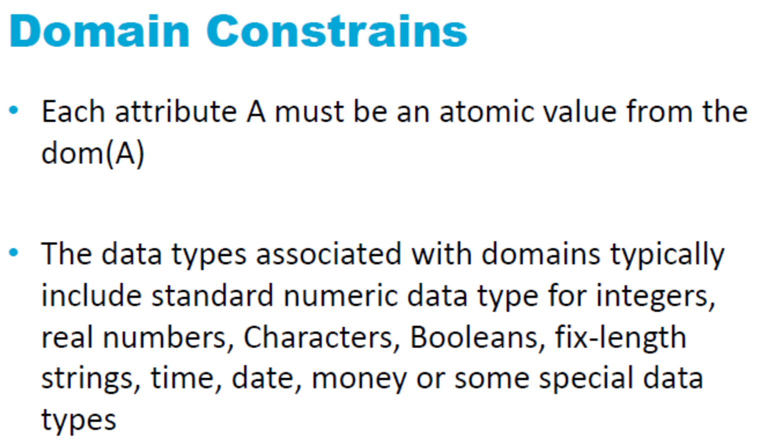
- **NOT NULL Constraint**
  - Remember, a relation is defined as $R \subseteq D_1 \times \ldots \times D_n$ with tuples $t \in R$
  - However, in a practical application its common that not always all attribute values are known
    - Therefore, it is usually assumed that there is a special NULL value in each domain, i.e. $NULL \in D_i$
  - Sometimes, this is not desired for certain attributes
    - Introduces the NOT NULL constraint
  - **Primary Key must never be NULL**
  - e.g. Address(street: string NOT NULL,
    number: numeric NOT NULL,
    zip-code: numeric NOT NULL,
    city: string NOT NULL
    postbox : string)



- **Foreign Key Constraint**
  - Sometimes, we want to link tuples in different relations
    - This will be integral for realizing ER relationships in a database
  - A **foreign key constraint** can be defined between the key attributes of one relation and some attributes of another one
    - e.g., Hero(id, first name, last name) ← References the key 'id' of hero
      Aliases(alias, heroid→Hero)
  - Tuples of the referring relation can only have values for the referencing attribute which are the key of an existing tuple in the referenced relation
    - This is called **referential integrity**



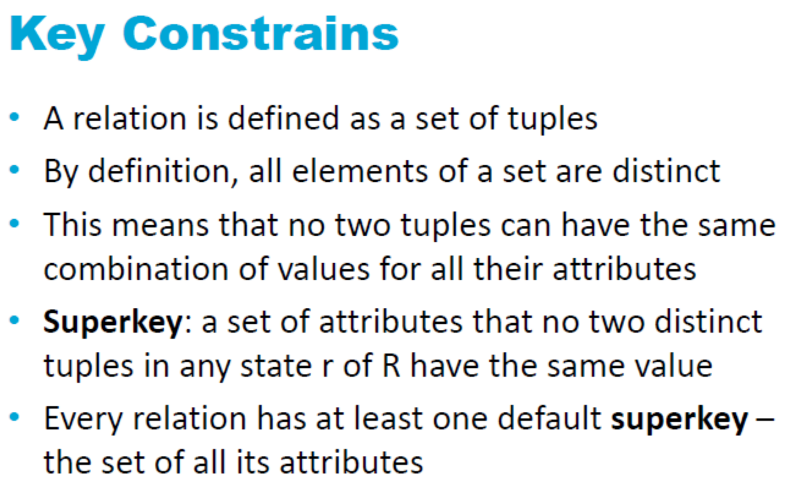
- Convention:
  - If a composite key is references, we write this as, e.g.,
    R1(a, b, c)   R2(d, e, f, (d, f)→R1)
  - This is not a standard notation, but rather close to what you find in SQL



Domain Constrains

- Each attribute A must be an atomic value from the dom(A)

- The data types associated with domains typically include standard numeric data type for integers, real numbers, Characters, Booleans, fix-length strings, time, date, money or some special data types



Key Constrains

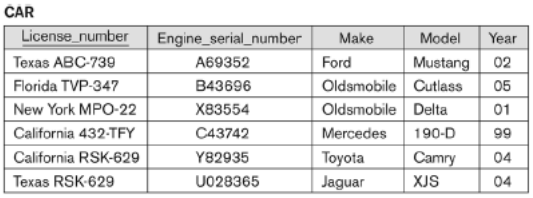
- A relation is defined as a set of tuples
- By definition, all elements of a set are distinct
- This means that no two tuples can have the same combination of values for all their attributes
- **Superkey**: a set of attributes that no two distinct tuples in any state r of R have the same value
- Every relation has at least one default **superkey** – the set of all its attributes

Key Constrains

- A superkey can have redundant attributes, so a more useful concept is that of a **KEY** which has no redundancy
- Key satisfied two constrains:
  - Two distinct tuple in any state of the relation cannot have identical values for the attributes in the key
  - It is a minimal superkey

- In general, a relation schema may have more than one key, in this case, each of the key is called a candidate key

CAR table with two candidate keys – LicenseNumber chosen as Primary Key



CAR

License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Figure 5.4
The CAR relation, with two candidate keys: License_number and Engine_serial_number.

Key Constrains

- If a relation has several **candidate keys**, one is chosen arbitrarily to be the **primary key**.

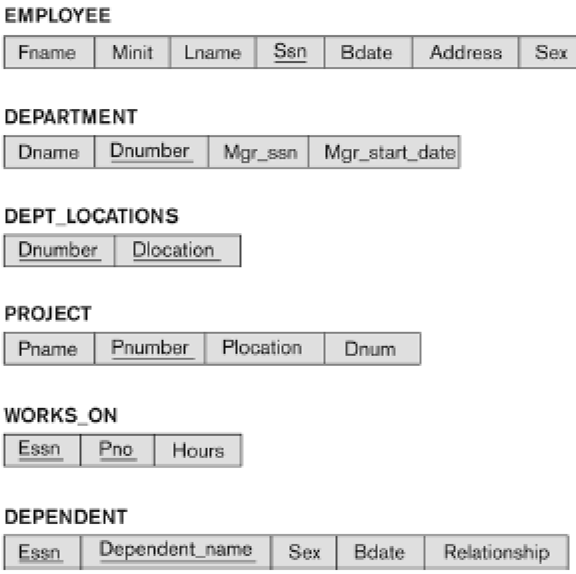
- Example: Consider the CAR relation schema:
  - CAR(State, Reg#, <u>SerialNo</u>, Make, Model, Year)
  - We chose SerialNo as the primary key

- The primary key value is used to *uniquely identify* each tuple in a relation
  - Provides the tuple identity

- Also used to ***reference*** **the tuple from another tuple**
  - General rule: Choose as primary key the smallest of the candidate keys (in terms of size)
  - Not always applicable – choice is sometimes subjective

Relational Database Schema

- **Relational Database Schema:**
  - A set S of relation schemas that belong to the same database.
  - S is the name of the whole **database schema**
  - S = {R1, R2, ..., Rn}
  - R1, R2, ..., Rn are the names of the individual **relation schemas** within the database S



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS

Dnumber	Dlocation

PROJECT

Pname	Pnumber	Plocation	Dnum

WORKS_ON

Essn	Pno	Hours

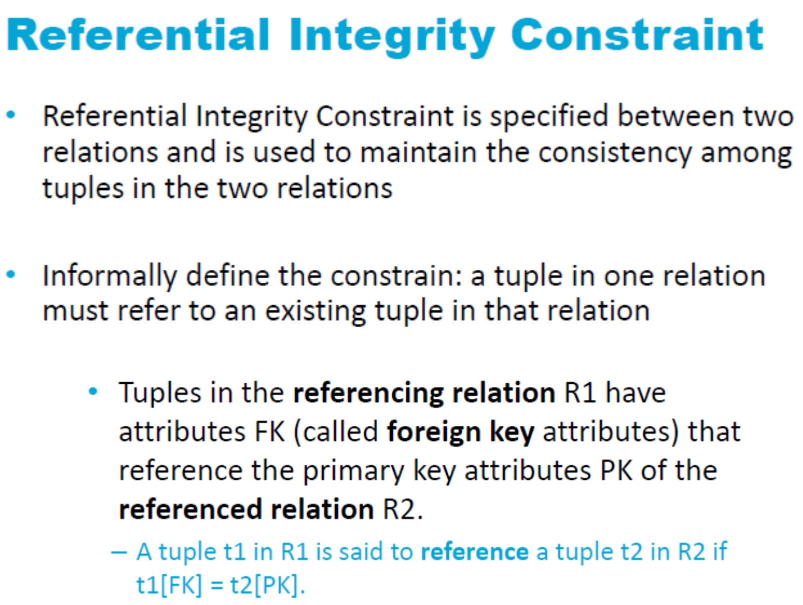
DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

Figure 5.5
Schema diagram for the COMPANY relational database schema.

Entity Integrity

- **Entity Integrity:**
  - The *primary key attributes* (PK) of each relation schema R in S cannot have null values in any tuple of r(R).
    - This is because primary key values are used to *identify* the individual tuples.
    - t[PK] ≠ null for any tuple t in r(R)
    - If PK has several attributes, null is not allowed in any of these attributes
  - Note: Other attributes of R may be constrained to disallow null values, even though they are not members of the primary key.

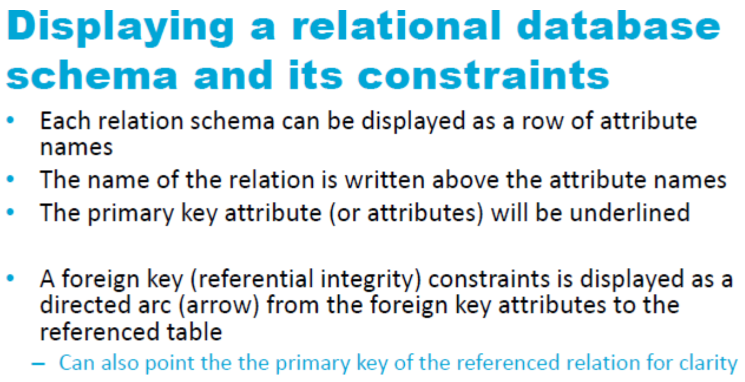


Referential Integrity Constraint

- Referential Integrity Constraint is specified between two relations and is used to maintain the consistency among tuples in the two relations

- Informally define the constrain: a tuple in one relation must refer to an existing tuple in that relation

  - Tuples in the **referencing relation** R1 have attributes FK (called **foreign key** attributes) that reference the primary key attributes PK of the **referenced relation** R2.
    - A tuple t1 in R1 is said to **reference** a tuple t2 in R2 if t1[FK] = t2[PK].



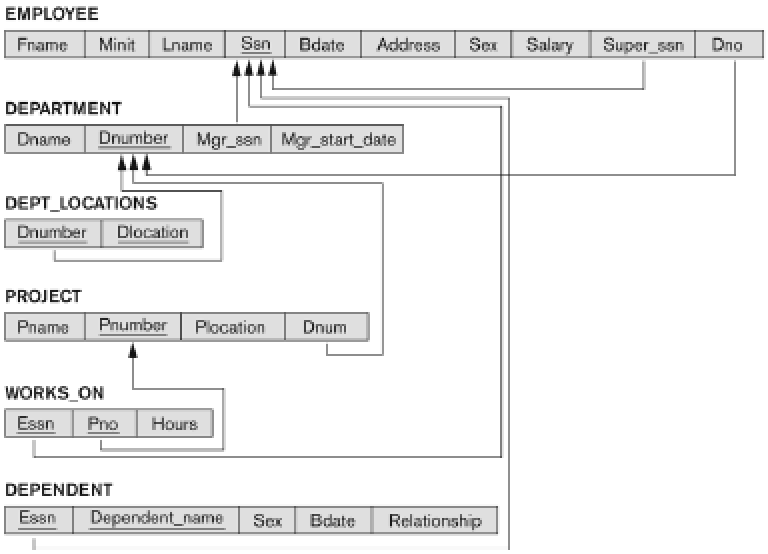
Displaying a relational database schema and its constraints

- Each relation schema can be displayed as a row of attribute names
- The name of the relation is written above the attribute names
- The primary key attribute (or attributes) will be underlined

- A foreign key (referential integrity) constraints is displayed as a directed arc (arrow) from the foreign key attributes to the referenced table
  - Can also point the the primary key of the referenced relation for clarity

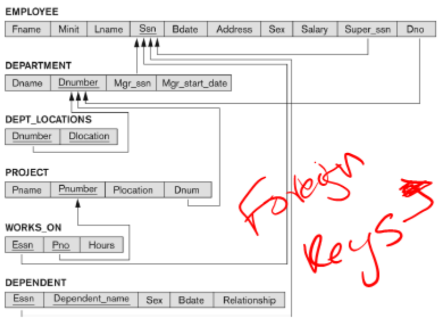
Referential Integrity Constraints for COMPANY database

Figure 5.7
Referential integrity constraints displayed on the COMPANY relational database schema.



Referential Integrity Constraints for COMPANY database

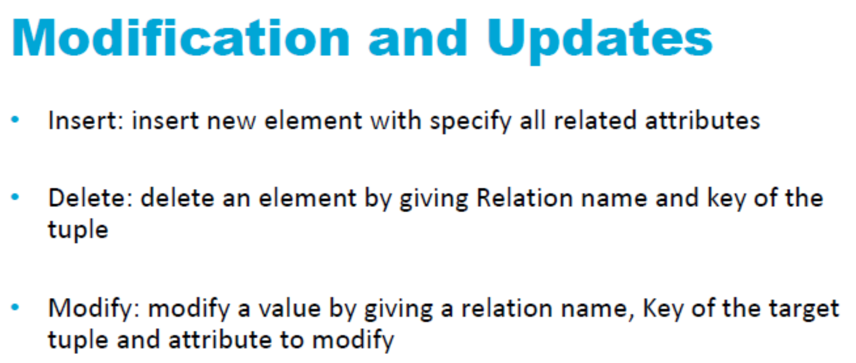
Figure 5.7
Referential integrity constraints displayed on the COMPANY relational database schema.



Other Types of Constraints

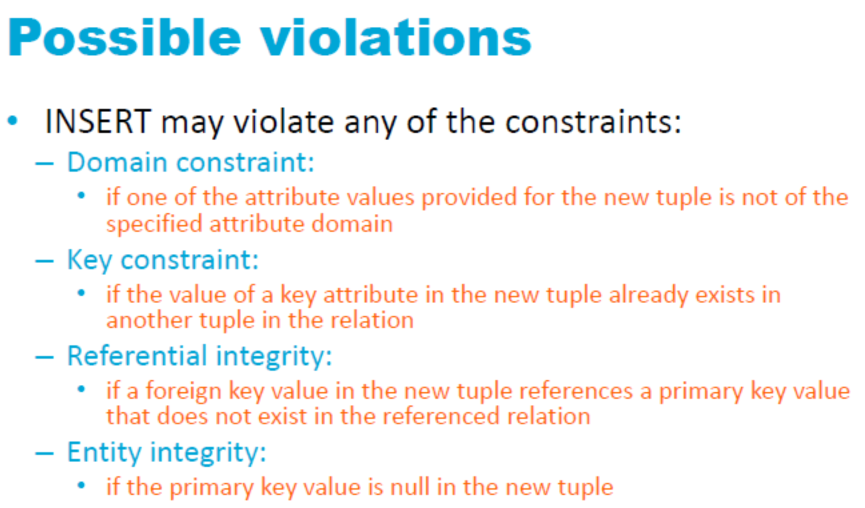
- Semantic Integrity Constraints:
  - based on application semantics and cannot be expressed by the model per se
  - Example: “the max. no. of hours per employee for all projects he or she works on is 56 hrs per week”

- A **constraint specification** language may have to be used to express these



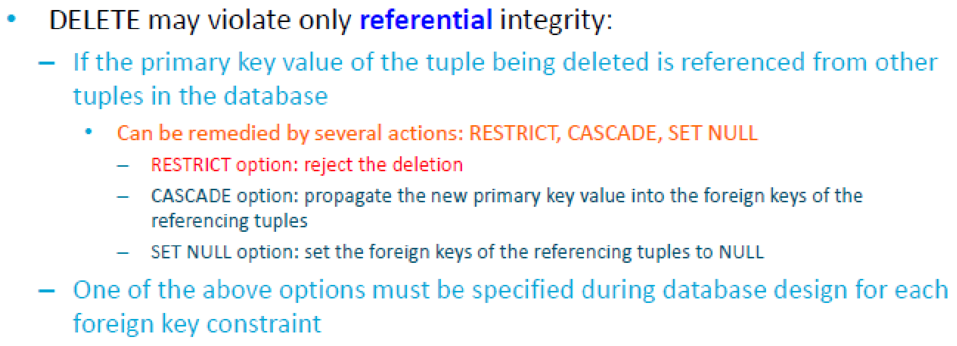
Modification and Updates

- Insert: insert new element with specify all related attributes
- Delete: delete an element by giving Relation name and key of the tuple
- Modify: modify a value by giving a relation name, Key of the target tuple and attribute to modify

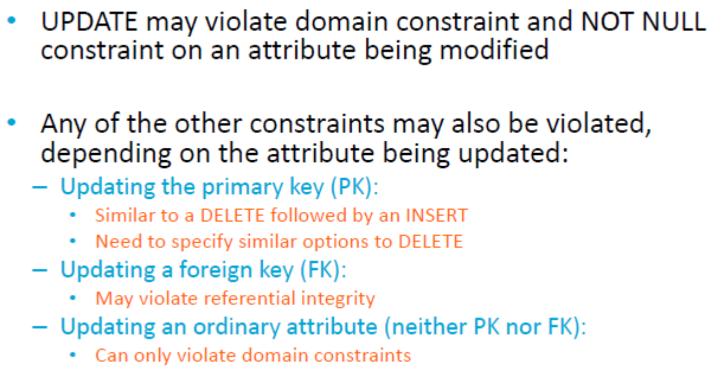


Possible violations

- INSERT may violate any of the constraints:
  - Domain constraint:
    - if one of the attribute values provided for the new tuple is not of the specified attribute domain
  - Key constraint:
    - if the value of a key attribute in the new tuple already exists in another tuple in the relation
  - Referential integrity:
    - if a foreign key value in the new tuple references a primary key value that does not exist in the referenced relation
  - Entity integrity:
    - if the primary key value is null in the new tuple

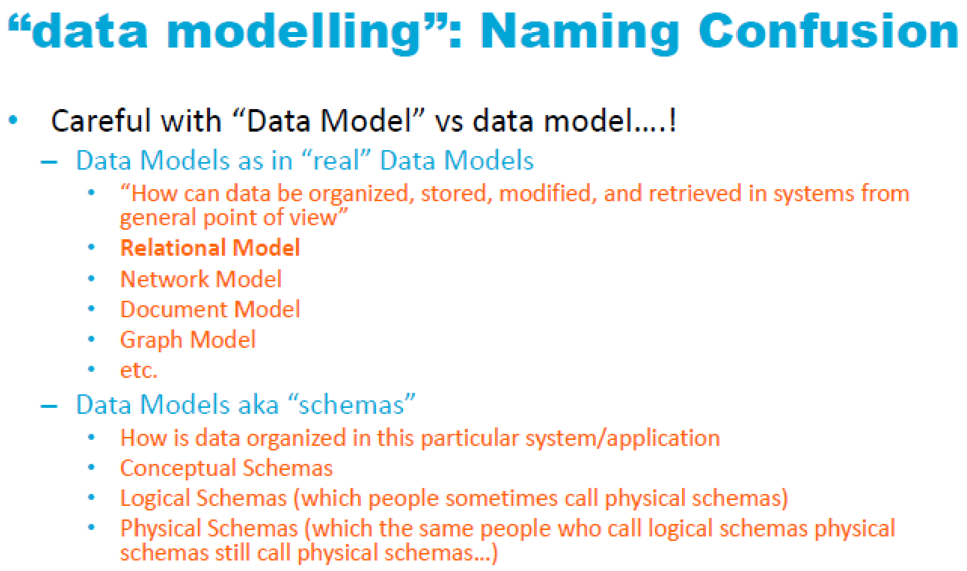


- DELETE may violate only **referential** integrity:
  - If the primary key value of the tuple being deleted is referenced from other tuples in the database
    - Can be remedied by several actions: RESTRICT, CASCADE, SET NULL
      - RESTRICT option: reject the deletion
      - CASCADE option: propagate the new primary key value into the foreign keys of the referencing tuples
      - SET NULL option: set the foreign keys of the referencing tuples to NULL
  - One of the above options must be specified during database design for each foreign key constraint



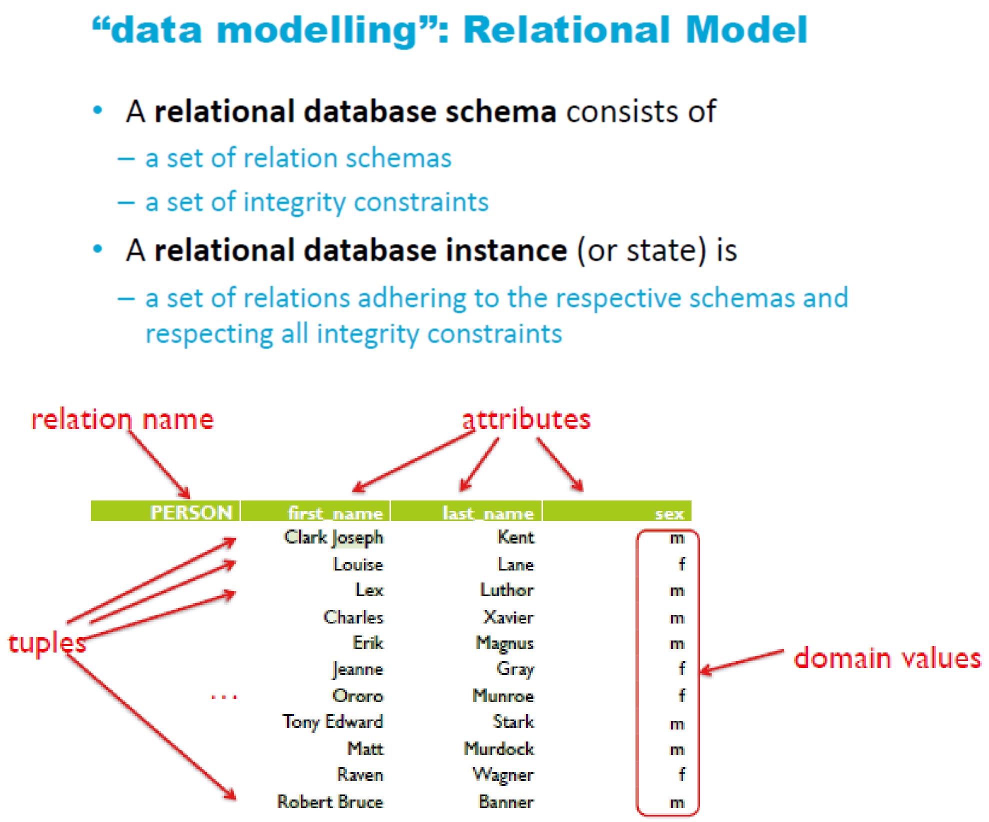
- UPDATE may violate domain constraint and NOT NULL constraint on an attribute being modified
- Any of the other constraints may also be violated, depending on the attribute being updated:
  - Updating the primary key (PK):
    - Similar to a DELETE followed by an INSERT
    - Need to specify similar options to DELETE
  - Updating a foreign key (FK):
    - May violate referential integrity
  - Updating an ordinary attribute (neither PK nor FK):
    - Can only violate domain constraints

Lecture 4. More on Modelling



“data modelling”: Naming Confusion

- Careful with “Data Model” vs data model….!
  - Data Models as in “real” Data Models
    - “How can data be organized, stored, modified, and retrieved in systems from general point of view”
    - **Relational Model**
    - Network Model
    - Document Model
    - Graph Model
    - etc.
  - Data Models aka “schemas”
    - How is data organized in this particular system/application
    - Conceptual Schemas
    - Logical Schemas (which people sometimes call physical schemas)
    - Physical Schemas (which the same people who call logical schemas physical schemas still call physical schemas…)



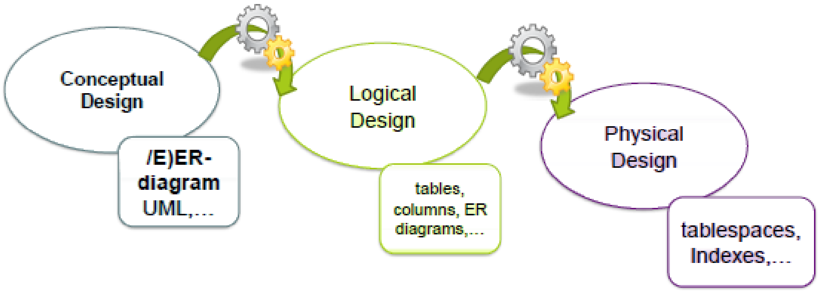
“data modelling”: Relational Model

- A **relational database schema** consists of
  - a set of relation schemas
  - a set of integrity constraints
- A **relational database instance** (or state) is
  - a set of relations adhering to the respective schemas and respecting all integrity constraints

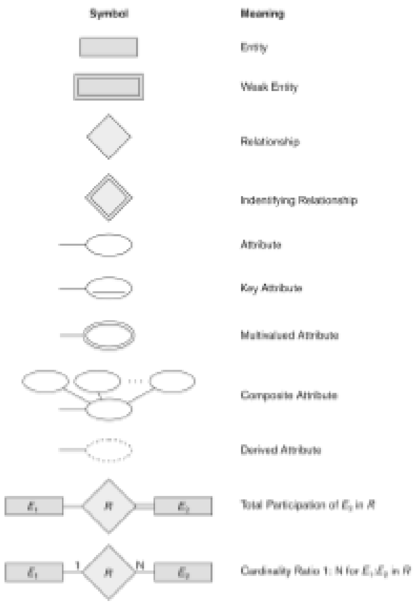
PERSON	first_name	last_name	sex
	Clark Joseph	Kent	m
	Louise	Lane	f
	Lex	Luthor	m
	Charles	Xavier	m
	Erik	Magnus	m
	Jeanne	Gray	f
…	Ororo	Munroe	f
	Tony Edward	Stark	m
	Matt	Murdock	m
	Raven	Wagner	f
	Robert Bruce	Banner	m

“data modelling”: Schema Design

- Modeling the data involves three design phases
  - result of one phase is input of the next phase
  - often, automatic transition is possible with some additional designer feedback



Conceptual Modelling: ER

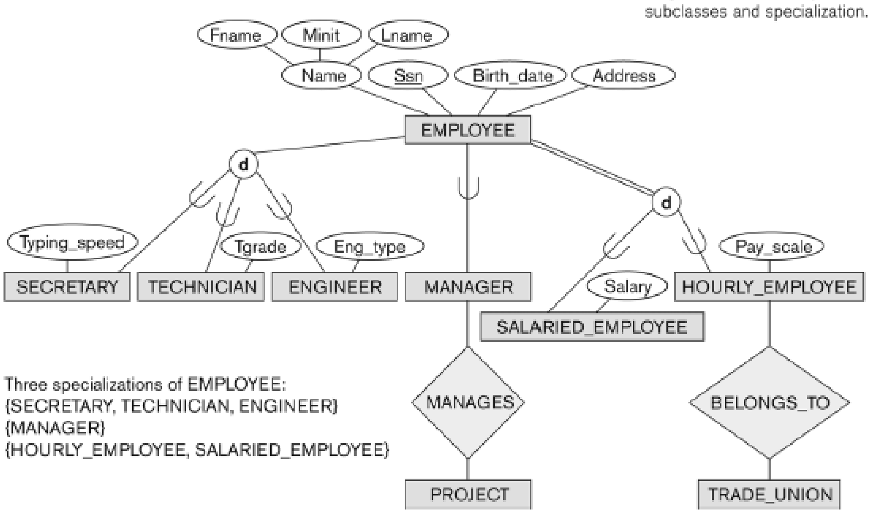


Cardinalities:

One: 1, I, ||, …

Multiple: *, N, M

Conceptual Modelling: EER

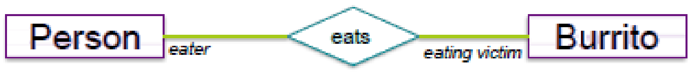


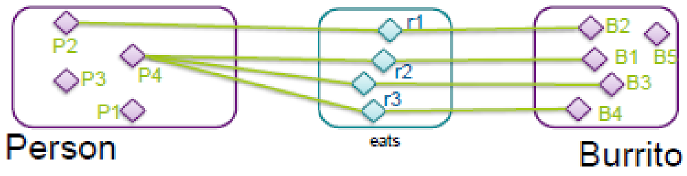
Three specializations of EMPLOYEE:
{SECRETARY, TECHNICIAN, ENGINEER}
{MANAGER}
{HOURLY_EMPLOYEE, SALARIED_EMPLOYEE}

Figure 4.1
EER diagram notation to represent subclasses and specialization.

- EER: Extended Entity-Relationship Modeling
  - Adds inheritance and union types
- Supertypes and Subtypes
  - Overlapping and Disjoint
    - Entity can be of multiple subtypes, or of only one
  - Total or Partial
    - Total: Each entity must be of one of the subtypes

Cardinal Nightmare



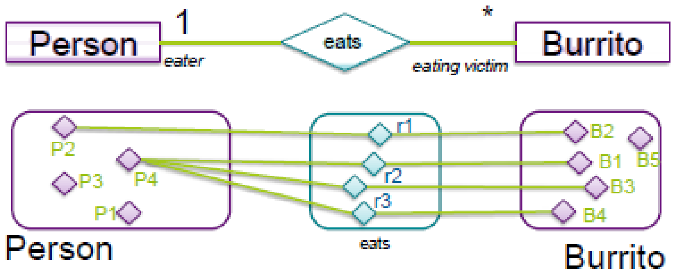


"N" and ""* means the same here

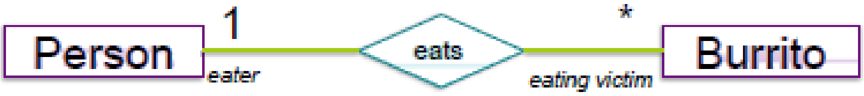
Cardinal Nightmare: UML style

- **UML Style** (our lecture's default):
  - Usually only distinguishes between 1, * or (0,1) or (1,*)
    - Fine-granular cardinalities and higher arity relations not easily possible

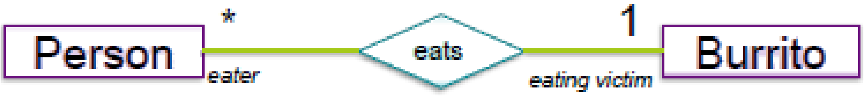
"One person eats multiple buritos"



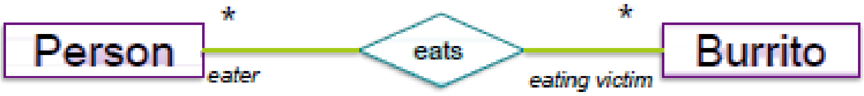
"One person eats multiple burritos."
(but there can be burritos being left over, or people who don't eat anything, or just 1, or 10)



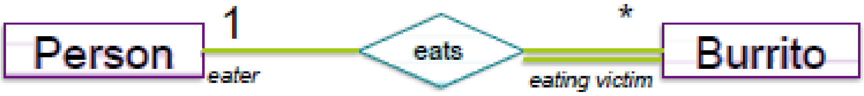
"Multiple persons share one burrito."
(but there can be burritos being left over, or people who don't eat anything)



"People eat burritos."
(Persons can share burritos, eat them by themselves, leave some over, or not eat anything)

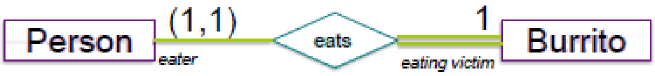


"One person eats multiple burritos."
(each burrito must be eaten, but some people can choose to not eat anything)



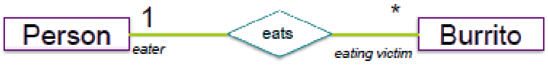
- UML style can also provide min / max
  - but only uses 0, 1, N/*
  - “1” stands for “(0,1)”: up to one
  - “N” stands for “(0,N)”: any number of
  - “(1,1)”: exactly 1
    - which is the same as 1 with total participation…
  - “(1,N)”: at least 1
    - Which is the same as N with total participation

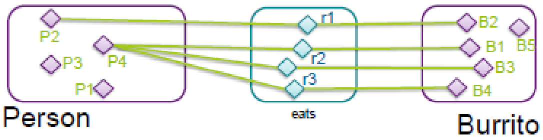
"Each person eats exactly one burtito, and each burrito is being eaten"



- **UML Style** (our lecture’s default):
  - Usually only distinguishes between 0, 0..1,1, 0..* and *
    - Fine-granular cardinalities and higher arity relations not easily possible

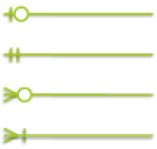
USE THIS IN THE EXAM





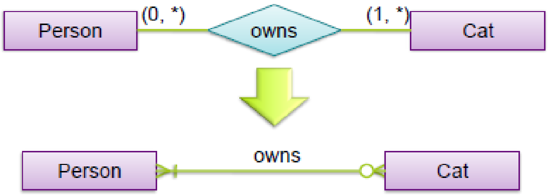
ER - Crow’s Foot Notation

- **Many** derivates of Chen ER became popular
- **Crow’s Foot Notation: Relationship Types**
  - relationship types are modeled by lines connecting the entities (no explicit symbol for relationships)
  - line is annotated with the name of the relationship which is a verb
  - cardinalities are represented graphically
    - **(0, 1):** zero or one
    - **(1, 1):** exactly one
    - **(0, *):** zero or more
    - **(1, *):** one or more



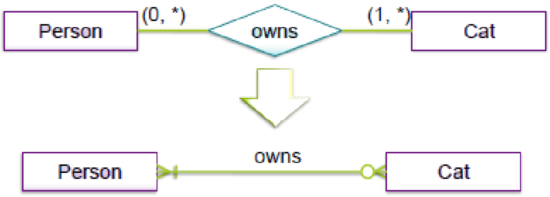
ER – Crow's Foot Notation

- Attention:
  - Cardinalities are written on the opposite side of the relationship (in contrast to Chen notation)



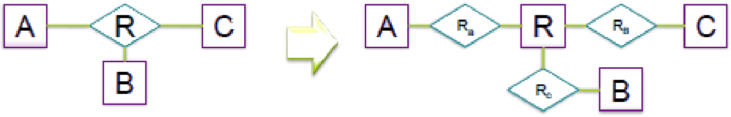
ER – Crow's Foot Notation

- Attention:
  - Cardinalities are written on the opposite side of the relationship (in contrast to Chen notation)

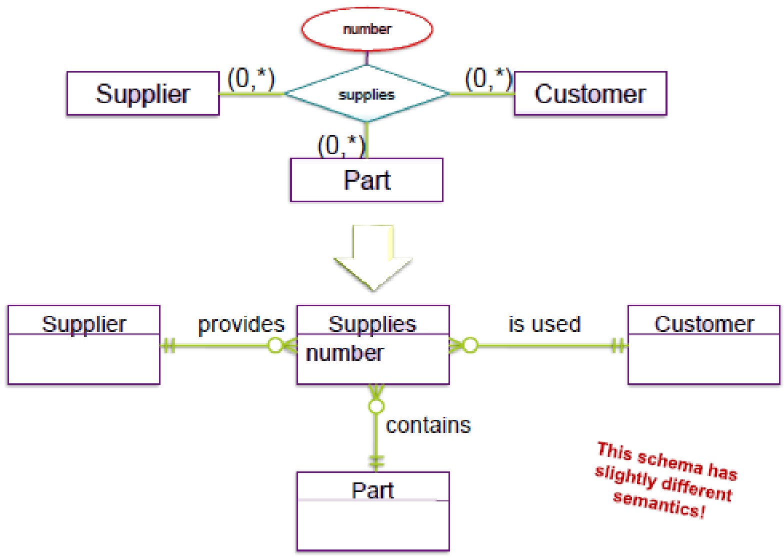


ER – Crow's Foot Notation

- **Problem**
  - N-ary relationship types **are not supported** by crow's foot notation, neither are relationship attributes
- **Workaround solution**:
  - **intermediate entities** must be used
    - N-ary relationships are broken down in a series of **binary** relationship types anchoring on the intermediate entity



ER – Crow’s Foot Notation



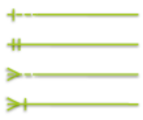
This schema has slightly different semantics!

ER – Crow’s Foot Notation

- Originally, ER diagrams were intended to be used on a **conceptual** level
  - model data in an abstract fashion **independent** of implementation
- Crow’s foot notation sacrifices some conceptual expressiveness
  - model is closer to the **logical** model (i.e. the way the data is later really stored in a system)
  - this is **not** always **desirable** and may obfuscate the intended semantics of the model

ER – Even more notations…

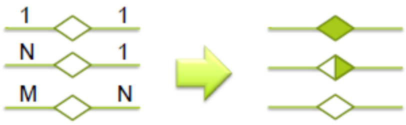
- **Barker’s notation**
  - based on Crow’s Foot Notation
  - developed by Richard Barker for **Oracle**’s CASE modeling books and tools in1986
  - cardinalities are represented differently
    - **(0, 1):** zero or one
    - **(1, 1):** exactly one
    - **(0, N):** zero or more
    - **(1, N):** one or more



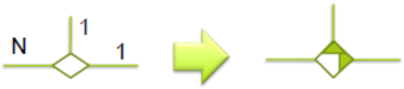
    - cardinalities position similar to Crow’s Foot notation and opposite to classic ER
  - different notation of subtypes

ER – Even more notations…

- **Black Diamond Notation**
  - cardinalities are represented differently
    - cardinality annotation per relationship, not per relationship end
    - 1:1
    - 1:N
    - N:M

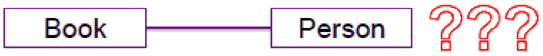


  - also, N-ary relationships possible
    - ternary

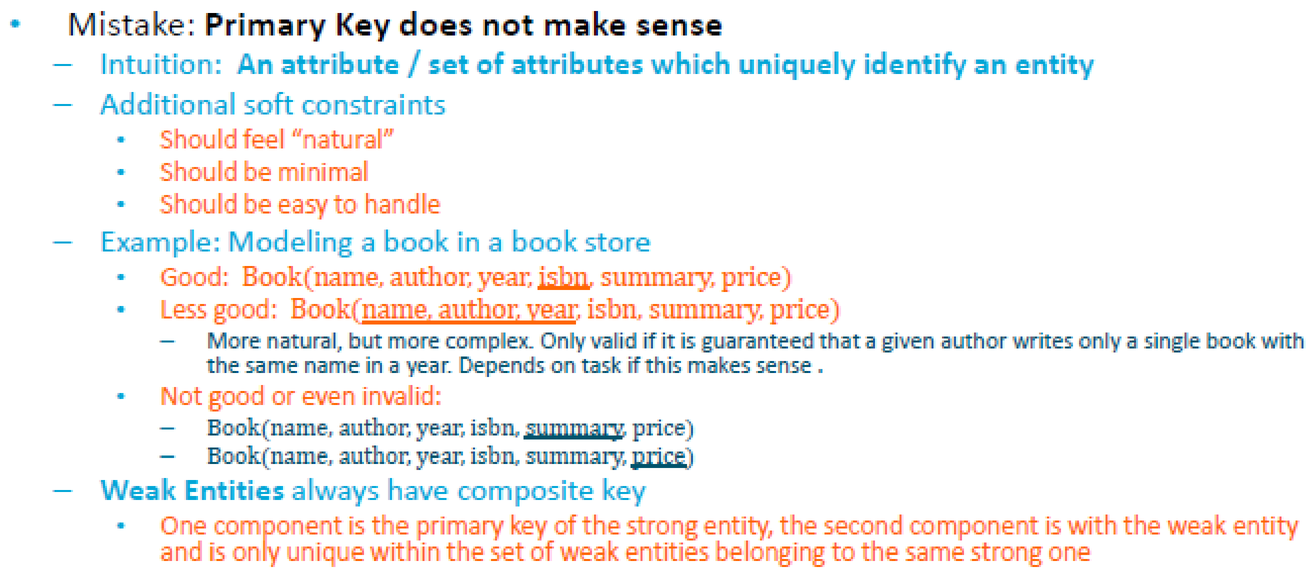


Common Mistakes in ER-Modelling

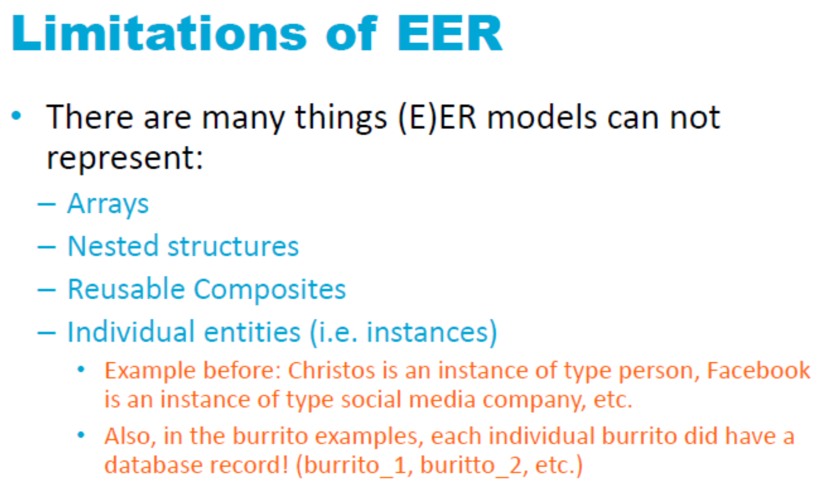
- Mistake: **No Primary Key**
  - Just don’t do that outside of simple examples
  - (same is true for cardinalities)
- Mistake: **No Relation Symbol or Name**



- Mistake: **Modelling Functionality as Data**
  - Mistake: **Model not suitable for the task**



- Mistake: **Primary Key does not make sense**
  - Intuition: **An attribute / set of attributes which uniquely identify an entity**
  - Additional soft constraints
    - Should feel "natural"
    - Should be minimal
    - Should be easy to handle
  - Example: Modeling a book in a book store
    - Good: Book(name, author, year, isbn, summary, price)
    - Less good: Book(name, author, year, isbn, summary, price)
      - More natural, but more complex. Only valid if it is guaranteed that a given author writes only a single book with the same name in a year. Depends on task if this makes sense .
    - Not good or even invalid:
      - Book(name, author, year, isbn, summary, price)
      - Book(name, author, year, isbn, summary, price)
  - **Weak Entities** always have composite key
    - One component is the primary key of the strong entity, the second component is with the weak entity and is only unique within the set of weak entities belonging to the same strong one

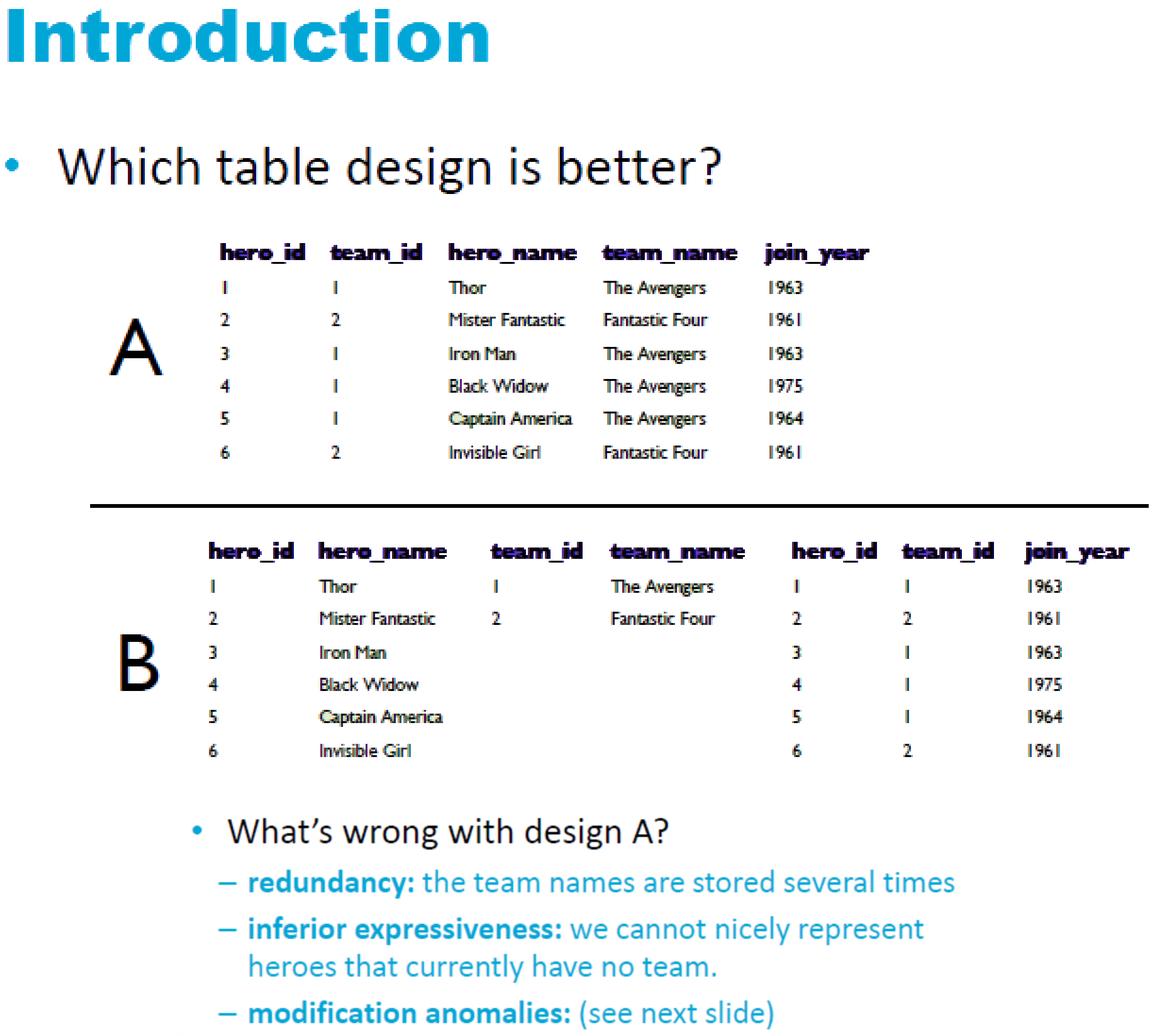


Limitations of EER

- There are many things (E)ER models can not represent:
  - Arrays
  - Nested structures
  - Reusable Composites
  - Individual entities (i.e. instances)
    - Example before: Christos is an instance of type person, Facebook is an instance of type social media company, etc.
    - Also, in the burrito examples, each individual burrito did have a database record! (burrito_1, buritto_2, etc.)

Week 3. Functional Dependencies and Normalization

Lecture 5. Functional Dependencies and Normalization



Introduction

- Which table design is better?

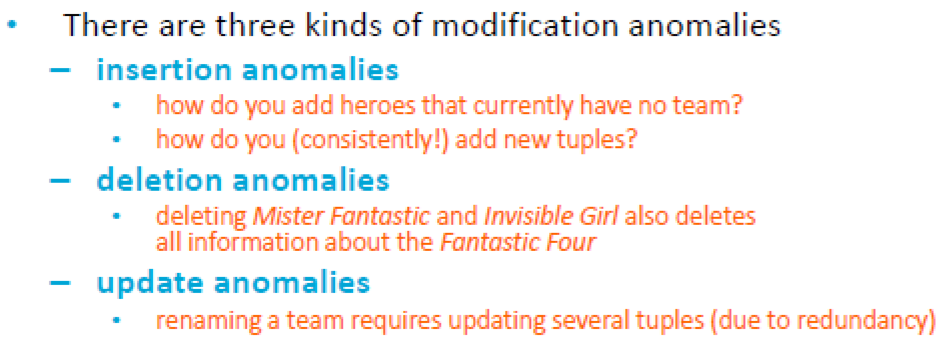
A

hero_id	team_id	hero_name	team_name	join_year
1	1	Thor	The Avengers	1963
2	2	Mister Fantastic	Fantastic Four	1961
3	1	Iron Man	The Avengers	1963
4	1	Black Widow	The Avengers	1975
5	1	Captain America	The Avengers	1964
6	2	Invisible Girl	Fantastic Four	1961

B

hero_id	hero_name	team_id	team_name	hero_id	team_id	join_year
1	Thor	1	The Avengers	1	1	1963
2	Mister Fantastic	2	Fantastic Four	2	2	1961
3	Iron Man			3	1	1963
4	Black Widow			4	1	1975
5	Captain America			5	1	1964
6	Invisible Girl			6	2	1961

- What's wrong with design A?
  - **redundancy:** the team names are stored several times
  - **inferior expressiveness:** we cannot nicely represent heroes that currently have no team.
  - **modification anomalies:** (see next slide)



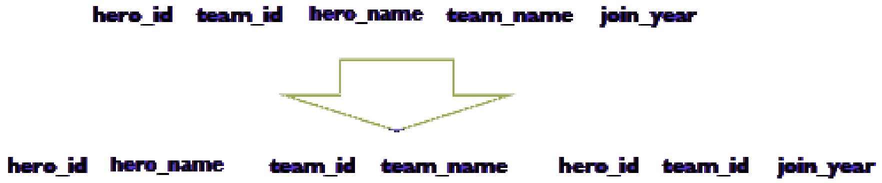
- There are three kinds of modification anomalies
  - **insertion anomalies**
    - how do you add heroes that currently have no team?
    - how do you (consistently!) add new tuples?
  - **deletion anomalies**
    - deleting *Mister Fantastic* and *Invisible Girl* also deletes all information about the *Fantastic Four*
  - **update anomalies**
    - renaming a team requires updating several tuples (due to redundancy)

- In general, ***good* relational database designs** have the following properties
  - redundancy is minimized
    - i.e. no information is represented several times!
    - logically distinct information is placed in distinct relation schemes
      - Do not misunderstand this as “never ever ever have any redundancy”
  - modification anomalies are prevented *by design*
    - i.e. by using keys and foreign keys, not by enforcing an excessive amount of (hard to check) constraints!
  - in practice, *good* designs should also match the characteristics of the used RDBMS
    - enable efficient query processing
    - ….this, however, might in some cases mean that redundancy is beneficial
      - It’s quite tricky to find the proper balance between different optimization goals
- It’s all about splitting up tables …
  - remember design B

Towards Normalization

- The *rules of thumb* for *good database design* can be formalized by the concept of relational database **normalization**
- But before going into details, let’s recap some definitions from the relational model
  - data is represented using a **relation schema** $S(R_1, ..., R_n)$
    - each relation $R(A_1, ... A_n)$ contains attributes $A_1, ... A_n$
  - a **relational database schema** consists of
    - a set of relations
    - a set of **integrity constraints** (e.g. *hero_id is unique* and *hero_id determines hero_name*)
  - a **relational database instance** (or extension) is
    - a set of tuples adhering to the respective schemas and respecting all integrity constraints

- For this lecture, let’s assume the following
  - $S(R_1, ..., R_n)$ is a **relation schema**
  - $R(A_1, ..., A_n)$ is a **relation** in S
  - $\mathcal{C}$ is a **set of constraints** satisfied by all extensions of S
- Our ultimate goal is to enhance the database design by **decomposing** the relations in *S* into a set of smaller relations, as we did in our example:


hero_id team_id hero_name team_name join_year
hero_id hero_name
team_id team_name
hero_id team_id join_year

Normalization

- **Definition (decomposition)**
  - let $\alpha_1, \ldots, \alpha_k \subseteq \{A_1, \ldots, A_n\}$ be k subsets of R's attributes
    - note that these subsets may be overlapping
  - then, for any α_i, a new relation R_i can be derived:

$$R_i = \pi_{\alpha_i}(R)$$

  - $\alpha_1, \ldots, \alpha_k$ is called a **decomposition** of R
- *Good* decompositions have to be **reversible**
  - the decomposition $\alpha_1, \ldots, \alpha_k$ is called **lossless** if and only if $R = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$, for any extension of R satisfying the constraints $\mathcal{C}$

$\mathcal{C}$ = {"{hero_id, team_id} is unique",
"hero_id determines hero_name",
"team_id determines team_name",
"{hero_id, team_id} determines join_year"}

Hero

hero_id	team_id	hero_name	team_name	join_year
1	1	Thor	The Avengers	1963
2	2	Mister Fantastic	Fantastic Four	1961
3	1	Iron Man	The Avengers	1963
4	1	Hulk	The Avengers	1963
5	1	Captain America	The Avengers	1964
6	2	Invisible Girl	Fantastic Four	1961

our example decomposition is lossless

α_1 = {heroID, heroname}, α_2 = {teamID, teamName}, α_3 = {heroID, teamID, joinYear}

π_{α_1}(Hero)

hero_id	hero_name
1	Thor
2	Mister Fantastic
3	Iron Man
4	Hulk
5	Captain America
6	Invisible Girl

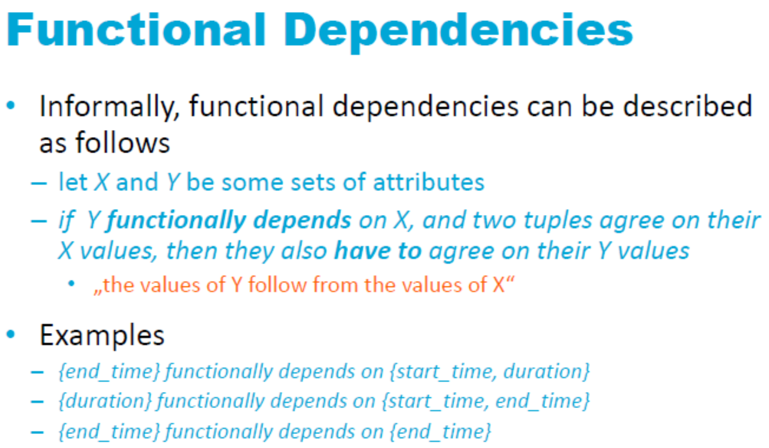
π_{α_2}(Hero)

team_id	team_name
1	The Avengers
2	Fantastic Four

π_{α_3}(Hero)

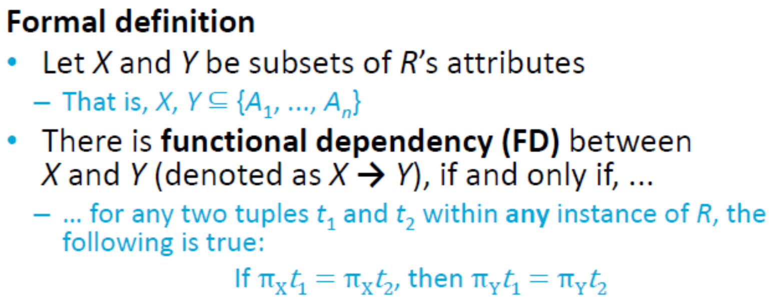
hero_id	team_id	join_year
1	1	1963
2	2	1961
3	1	1963
4	1	1963
5	1	1964
6	2	1961

- **Normalizing** a relation schema S means replacing relations in S by lossless decompositions
- However, this raises some new questions
  - under which conditions is there a (nontrivial) lossless decomposition?
    - decompositions involving $\alpha_i = \{A_1, \ldots, A_n\}$ or $\alpha_i = \emptyset$ are called **trivial**
  - if there is a lossless decomposition, how to find it?
  - how to measure a relation schema's *design quality*?
    - We may abstain from further normalization if the quality is *good enough*…



Functional Dependencies

- Informally, functional dependencies can be described as follows
  - let *X* and *Y* be some sets of attributes
  - *if Y **functionally depends** on X, and two tuples agree on their X values, then they also **have to** agree on their Y values*
    - „the values of Y follow from the values of X“
- Examples
  - *{end_time} functionally depends on {start_time, duration}*
  - *{duration} functionally depends on {start_time, end_time}*
  - *{end_time} functionally depends on {end_time}*

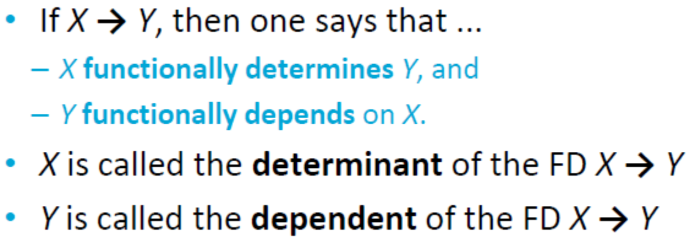


Formal definition

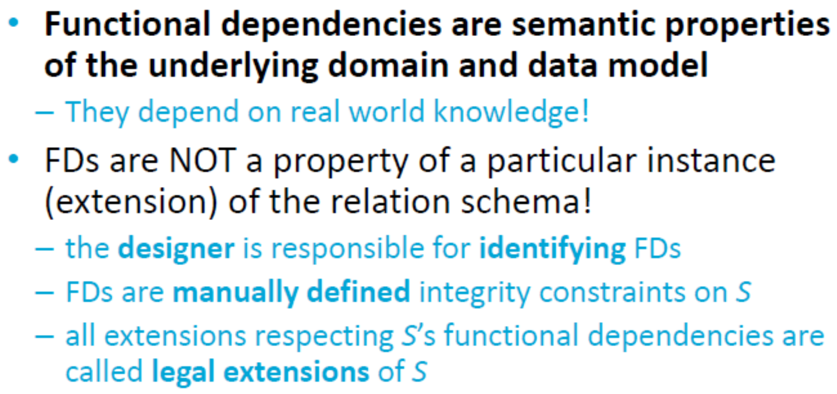
- Let *X* and *Y* be subsets of *R*'s attributes
  - That is, $X, Y \subseteq \{A_1, \ldots, A_n\}$
- There is **functional dependency (FD)** between *X* and *Y* (denoted as $X \rightarrow Y$), if and only if, ...
  - ... for any two tuples t_1 and t_2 within **any** instance of *R*, the following is true:

$$\text{If } \pi_X t_1 = \pi_X t_2, \text{ then } \pi_Y t_1 = \pi_Y t_2$$

(This means: if for two given tuples the attributes in X have the same value, then also the attributes Y need to have the same value.)



- If $X \rightarrow Y$, then one says that ...
  - *X* **functionally determines** *Y*, and
  - *Y* **functionally depends** on *X*.
- *X* is called the **determinant** of the FD $X \rightarrow Y$
- *Y* is called the **dependent** of the FD $X \rightarrow Y$

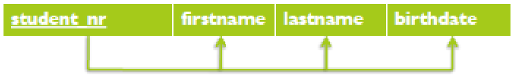


- **Functional dependencies are semantic properties of the underlying domain and data model**
  - They depend on real world knowledge!
- FDs are NOT a property of a particular instance (extension) of the relation schema!
  - the **designer** is responsible for **identifying** FDs
  - FDs are **manually defined** integrity constraints on *S*
  - all extensions respecting *S*'s functional dependencies are called **legal extensions** of *S*

- In fact, functional dependencies are a generalization of **key constraints**
- To show this, we need a short recap of **keys**
  - a set of attributes *X* is a **(candidate) key** for *R* if and only if it has both of the following properties
    - **uniqueness:** no legal instance of *R* ever contains two distinct tuples with the same value for *X*
    - **irreducibility:** no proper subset of *X* has the uniqueness property
  - a **superkey** is a superset of a key
    - i.e. only uniqueness is required

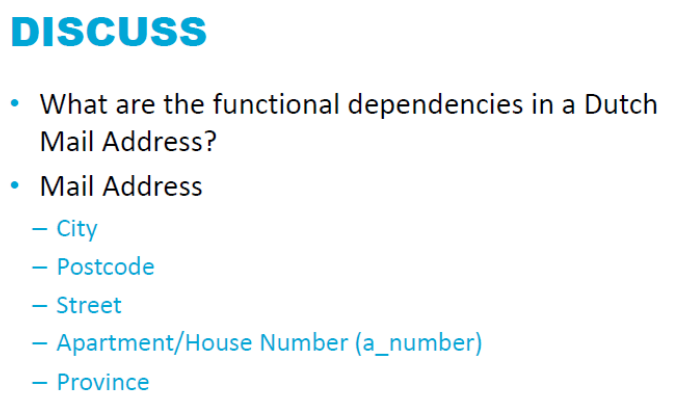
- In practice, if there is more than one candidate key, we usually choose one and call it the **primary key**
  - however, for normalization purposes, only candidate keys are important – thus, **we ignore primary keys** today
- The following is true
  - *X* is a superkey of *R*
    if and only if
    $X \to \{A_1, \ldots, A_n\}$ is a functional dependency in *R*

- Example
  - a relation containing students
    - semantics: student_nr is **unique**
    - {student_nr} → {firstname, lastname, birthdate}



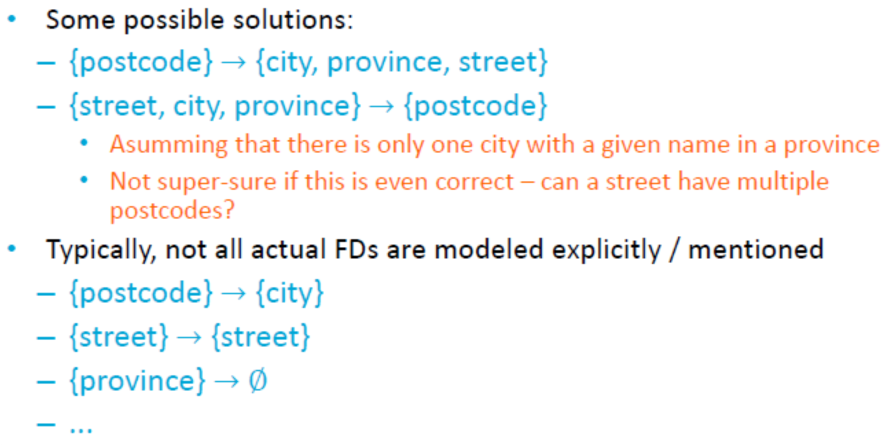
student_nr | firstname | lastname | birthdate

- Quick Summary on keys:
  - **Candidate Key** (or simply key)
    - A **irreducible** set of attributes which **uniquely** identifies a tuple
      - i.e.: all non-key attributes are functional dependent on the key, and no attribute can be removed without loosing the key properties
      - „Identifier“
  - **Superkey** is a superset of a candidate key
    - i.e. only uniqueness is required
      - Superkey also identifies a tuple, but is reducible
  - **Primary Key**
    - A primary key is one single key chosen from the set of candidate keys by the database designer
      - This choice impacts the way the DBMS manages relations and queries

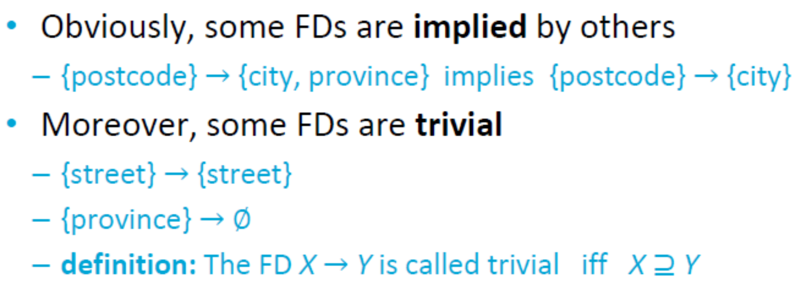


DISCUSS

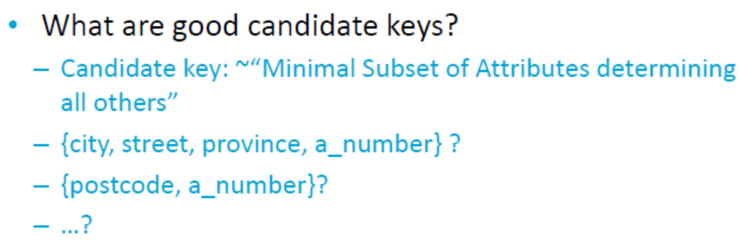
- What are the functional dependencies in a Dutch Mail Address?
- Mail Address
  - City
  - Postcode
  - Street
  - Apartment/House Number (a_number)
  - Province



- Some possible solutions:
  - {postcode} → {city, province, street}
  - {street, city, province} → {postcode}
    - Asumming that there is only one city with a given name in a province
    - Not super-sure if this is even correct – can a street have multiple postcodes?
- Typically, not all actual FDs are modeled explicitly / mentioned
  - {postcode} → {city}
  - {street} → {street}
  - {province} → ∅
  - …



- Obviously, some FDs are **implied** by others
  - {postcode} → {city, province} implies {postcode} → {city}
- Moreover, some FDs are **trivial**
  - {street} → {street}
  - {province} → ∅
  - **definition:** The FD $X \rightarrow Y$ is called trivial iff $X \supseteq Y$



- What are good candidate keys?
  - Candidate key: ~"Minimal Subset of Attributes determining all others"
  - {city, street, province, a_number} ?
  - {postcode, a_number}?
  - …?

Functional Dependencies

- **Definition:**
  For any set F of FDs, the **closure** of F (denoted F^+) is the set of all FDs that are logically **implied** by F
  - *Abstract Definition: F* **implies** $X \rightarrow Y$, if and only if any extension of R satisfying any FD in F, also satisfies the $X \rightarrow Y$

- Fortunately, the closure of F can easily be computed using a small set of **inference rules**

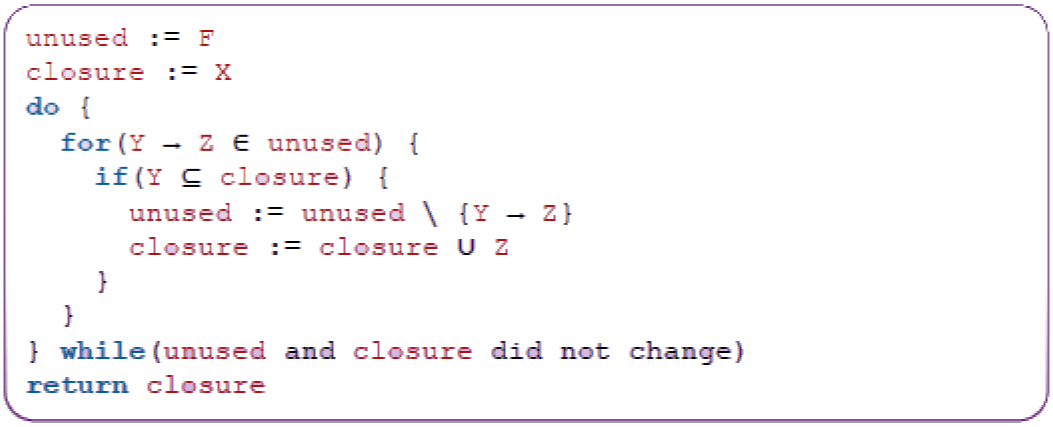
- For any attribute sets X, Y, Z, the following is true
  - **reflexivity:**
    If $X \supseteq Y$, then $X \rightarrow Y$
  - **augmentation:**
    If $X \rightarrow Y$, then $X \cup Z \rightarrow Y \cup Z$
  - **transitivity:**
    If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- These rules are called **Armstrong's axioms**
  - one can show that they are **complete** and **sound**
    - **completeness:** every implied FD can be derived
    - **soundness:** no non-implied FD can be derived

- To **simplify the practical task** of computing F^+ from F, several **additional rules** can be derived from Armstrong's axioms:
  - **decomposition:**
    If $X \rightarrow Y \cup Z$, then $X \rightarrow Y$ and $X \rightarrow Z$
  - **union:**
    If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow Y \cup Z$
  - **composition:**
    If $X \rightarrow Y$ and $Z \rightarrow W$, then $X \cup Z \rightarrow Y \cup W$

- Example
  - relational schema R(A, B, C, D, E, F)
  - FDs: {A} → {B, C} {B} → {E} {C, D} → {E, F}
  - then we can make the following derivation
    1. **{A} → {B, C}** (given)
    2. {A} → {C} (by decomposition)
    3. {A, D} → {C, D} (by augmentation)
    4. {A, D} → {E, F} (by transitivity with given {C, D} → {E, F})
    5. **{A, D} → {F}** (by decomposition)

- In principle, we can compute the closure F^+ of a given set F of FDs by means of the following algorithm:
  - *Repeatedly apply the six inference rules until they stop producing new FDs.*
- In practice, this algorithm is hardly very efficient
  - however, there usually is little need to compute the full closure
  - instead, it often suffices to compute a certain subset of the closure: the subset consisting of all FDs with given left side
    - This will later serve for finding proper keys or normalizing relations

- **Definition:**
  Given a set of attributes X and a set of FDs F,
  the **closure** of X under F, written as $(X, F)^+$,
  consists of all attributes that functionally depend on X
  - i.e. $(X, F)^+ := \{A_i \mid X \rightarrow A_i \text{ is implied by } F\}$
- The following algorithm computes $(X, F)^+$:



```
unused := F
closure := X
do {
  for(Y → Z ∈ unused) {
    if(Y ⊆ closure) {
      unused := unused \ {Y → Z}
      closure := closure ∪ Z
    }
  }
} while(unused and closure did not change)
return closure
```

- **Quiz**
  - $F = \{ \{A\} \rightarrow \{B, C\}, \{E\} \rightarrow \{C, G\},$
    $\{B\} \rightarrow \{E\}, \{C, D\} \rightarrow \{E, G\} \}$
  - *What is the closure of {A, B} under f?*

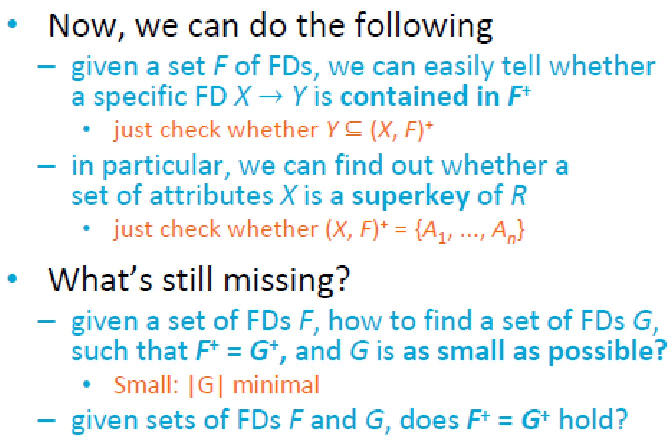
Intermediate Closure: {A, B}

Add C, because {A} → {B, C}   {A, B, C}

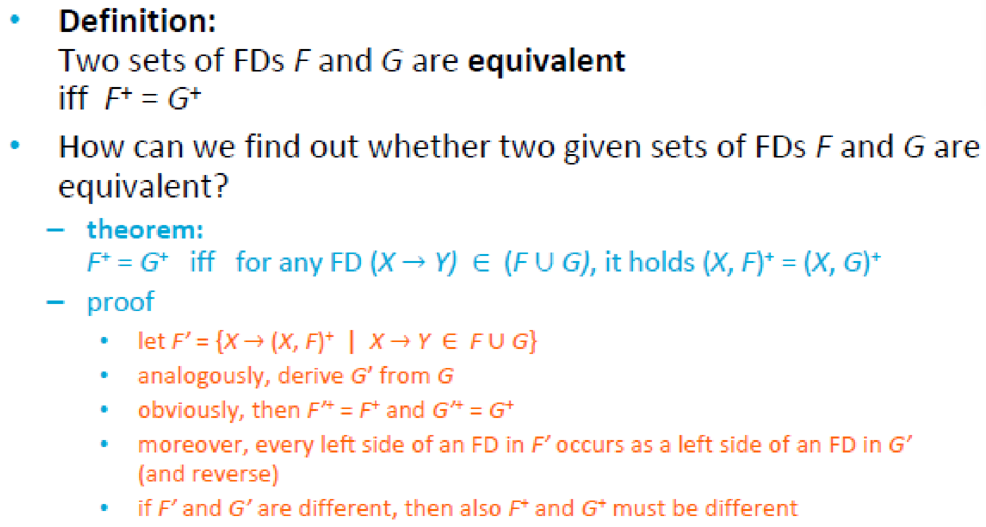
Add E, because {B} → {E}   {A, B, C, E}

Add F, because {E} → {C, G}

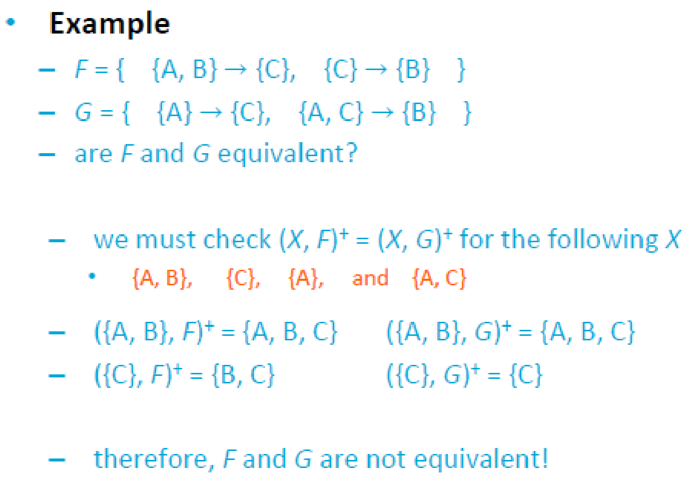
$(\{A, B\}, F)^+ = \{A, B, C, E, G\}$



- Now, we can do the following
  - given a set F of FDs, we can easily tell whether a specific FD $X \rightarrow Y$ is **contained in F^+**
    - just check whether $Y \subseteq (X, F)^+$
  - in particular, we can find out whether a set of attributes X is a **superkey** of R
    - just check whether $(X, F)^+ = \{A_1, \ldots, A_n\}$
- What's still missing?
  - given a set of FDs F, how to find a set of FDs G, **such that $F^+ = G^+$**, and G is **as small as possible?**
    - Small: |G| minimal
  - given sets of FDs F and G, **does $F^+ = G^+$** hold?



- **Definition:**
  Two sets of FDs F and G are **equivalent**
  iff $F^+ = G^+$
- How can we find out whether two given sets of FDs F and G are equivalent?
  - **theorem:**
    $F^+ = G^+$ iff for any FD $(X \rightarrow Y) \in (F \cup G)$, it holds $(X, F)^+ = (X, G)^+$
  - proof
    - let $F' = \{X \rightarrow (X, F)^+ \mid X \rightarrow Y \in F \cup G\}$
    - analogously, derive G' from G
    - obviously, then $F'^+ = F^+$ and $G'^+ = G^+$
    - moreover, every left side of an FD in F' occurs as a left side of an FD in G' (and reverse)
    - if F' and G' are different, then also F^+ and G^+ must be different



- **Example**
  - $F = \{ \{A, B\} \rightarrow \{C\}, \{C\} \rightarrow \{B\} \}$
  - $G = \{ \{A\} \rightarrow \{C\}, \{A, C\} \rightarrow \{B\} \}$
  - are F and G equivalent?
  - we must check $(X, F)^+ = (X, G)^+$ for the following X
    - {A, B}, {C}, {A}, and {A, C}
  - $(\{A, B\}, F)^+ = \{A, B, C\}$ $(\{A, B\}, G)^+ = \{A, B, C\}$
  - $(\{C\}, F)^+ = \{B, C\}$ $(\{C\}, G)^+ = \{C\}$
  - therefore, F and G are not equivalent!

- **Remember:**
  To have a **small representation** of *F*, we want to find a *G*, such that
  - *F* and *G* are equivalent
  - *G* is *as small as possible* (we will call this property **minimality**)

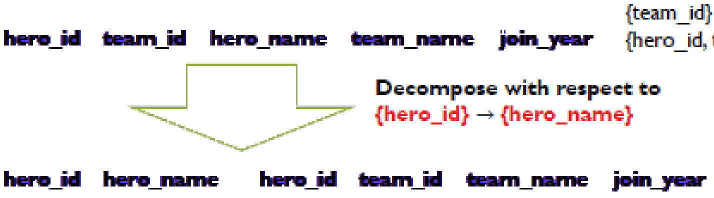
- **Definition:**
  A set of FDs *F* is **minimal** iff the following is true
  - every FD $X \to Y$ in *F* is **in canonical form**
    - i.e. *Y* consists of exactly one attribute
  - every FD $X \to Y$ in *F* is **left-irreducible**
    - i.e. no attribute can be removed from *X* without changing F^+
  - every FD $X \to Y$ in *F* is **non-redundant**
    - i.e. $X \to Y$ cannot be removed from *F* without changing F^+

- The following algorithm *minimizes* *F*, that is,
  it transforms *F* into a minimal equivalent of *F*
  1. Split up all right sides to get FDs in canonical form.
  2. Remove all redundant attributes from the left sides (by checking which attribute removals change F^+).
  3. Remove all redundant FDs from *F*
     (by checking which FD removals change F^+).

- **Example**
  - given *F* = {
    $\{A\} \to \{B, C\}$, $\{B\} \to \{C\}$,
    $\{A\} \to \{B\}$, $\{A, B\} \to \{C\}$,
    $\{A, C\} \to \{D\}$
    }
  1. Split up the right sides:
     $\{A\} \to \{B\}$, $\{A\} \to \{C\}$, $\{B\} \to \{C\}$,
     $\{A, B\} \to \{C\}$, $\{A, C\} \to \{D\}$
  2. Remove C from $\{A, C\} \to \{D\}$:
     - $\{A\} \to \{C\}$ implies $\{A\} \to \{A, C\}$ (augmentation)
     - $\{A\} \to \{A, C\}$ and $\{A, C\} \to \{D\}$ imply $\{A\} \to \{D\}$ (transitivity)
  - Now we have:
    $\{A\} \to \{B\}$, $\{A\} \to \{C\}$, $\{B\} \to \{C\}$,
    $\{A, B\} \to \{C\}$, $\{A\} \to \{D\}$
  3. Remove $\{A, B\} \to \{C\}$:
     - $\{A\} \to \{C\}$ implies $\{A, B\} \to \{C\}$
  4. Remove $\{A\} \to \{C\}$:
     - $\{A\} \to \{B\}$ and $\{B\} \to \{C\}$ imply $\{A\} \to \{C\}$ (transitivity)
  - Finally, we end up with a **minimal equivalent** of *F*:
    $\{A\} \to \{B\}$, $\{B\} \to \{C\}$, $\{A\} \to \{D\}$

- **Functional dependencies are the perfect tool for performing lossless decompositions**
  - **Heath's Theorem:**
    Let $X \rightarrow Y$ be an FD constraint of the relation schema $R(A_1, \ldots, A_n)$. Then, the following decomposition of R is **lossless:**
    $$\alpha_1 = X \cup Y \quad \text{and} \quad \alpha_2 = \{A_1, \ldots, A_n\} \setminus Y.$$
  - **Example:**

FDs:
{hero_id} → {hero_name}
{team_id} → {team_name}
{hero_id, team_id} → {join_year}



hero_id **team_id** **hero_name** **team_name** **join_year**

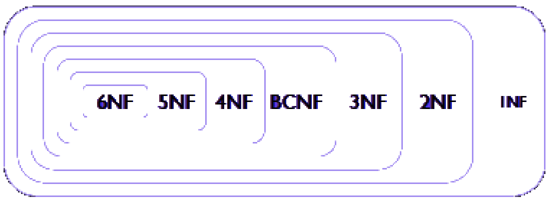
Decompose with respect to
{hero_id} → {hero_name}

hero_id **hero_name**   **hero_id** **team_id** **team_name** **join_year**

Normal Forms

- Back to normalization
  - remember:
    normalization = finding lossless decompositions
  - but only decompose, if the relation schema is of bad quality
- How to measure the quality of a relation schema?
  - claim: the quality depends on the constraints
  - in our case:
    quality depends on the FDs of the relation schema
  - schemas can be classified into different quality levels, which are called normal forms

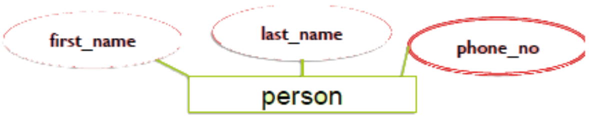
- Part of a schema design process is to choose a desired normal form and convert the schema into that form
- There are seven normal forms
  - the higher the number, …
    - … the stricter the requirements,
    - … the less anomalies and redundancy, and
    - … the better the *design quality*.



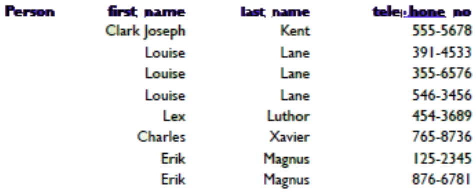
1NF Composite attributes to atoms

- **First normal form (1NF)**
  - has nothing to do with functional dependencies!
  - restricts relations to being *flat*
  - the value of any attribute must be **atomic,** that is, it **cannot be composed** of several other attributes
    - if this property is met, the relation is often referred to as a being in **first normal form** (1NF or minimal form)
    - in particular, **set-valued** and relation-valued attributes (tables within tables) are **prohibited**

1NF



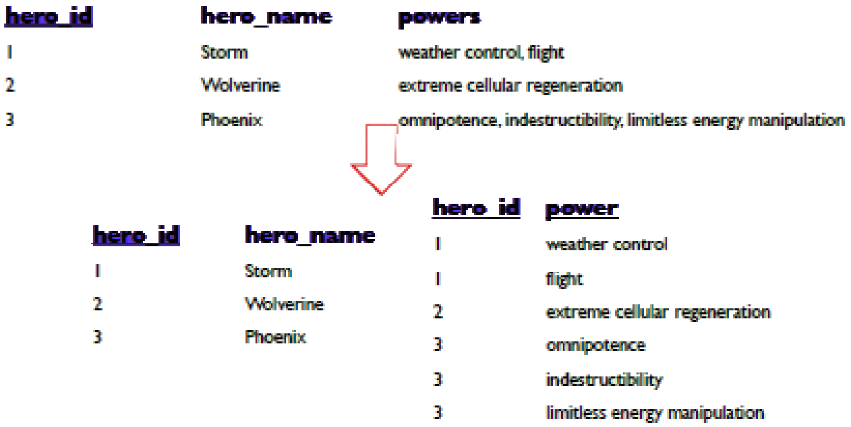
- Please note, it is possible to **model** composed attributes in ER models…
- To transform such a model into the relational model, a **normalization** step is needed
  - this is not always trivial, e.g., what happens to keys?



Person	first_name	last_name	telephone_no
	Clark Joseph	Kent	555-5678
	Louise	Lane	391-4533
	Louise	Lane	355-6576
	Louise	Lane	546-3456
	Lex	Luthor	454-3689
	Charles	Xavier	765-8736
	Erik	Magnus	125-2345
	Erik	Magnus	876-6781

- multi-valued attributes must be normalized, e.g., by
  - a) introducing a **new relation** for the multi-valued attribute
    - most common solution
  - b) **replicating** the tuple for each multi-value
    - as e.g., often done for song list metadata (e.g., mp3 tags)
  - c) introducing an **own attribute** for each multi-value (if there is a small maximum number of values)
    - as sometimes done in Big Data Database (e.g., Bigtable)

- a) Introducing a **new relation**
  - uses old key and multi-attribute as composite key

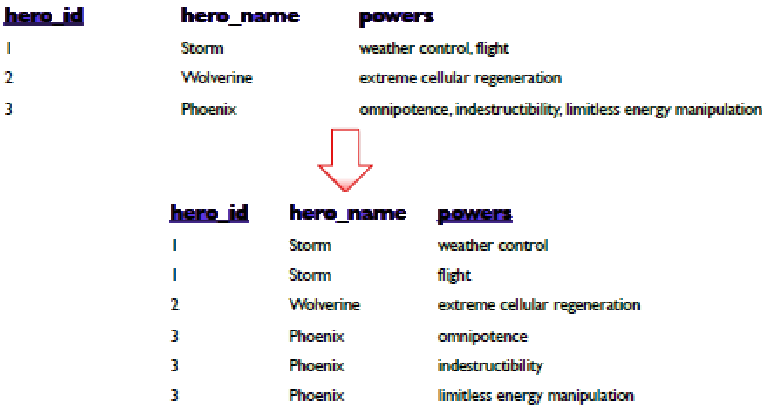


hero_id	hero_name	powers
1	Storm	weather control, flight
2	Wolverine	extreme cellular regeneration
3	Phoenix	omnipotence, indestructibility, limitless energy manipulation

hero_id	hero_name
1	Storm
2	Wolverine
3	Phoenix

hero_id	power
1	weather control
1	flight
2	extreme cellular regeneration
3	omnipotence
3	indestructibility
3	limitless energy manipulation

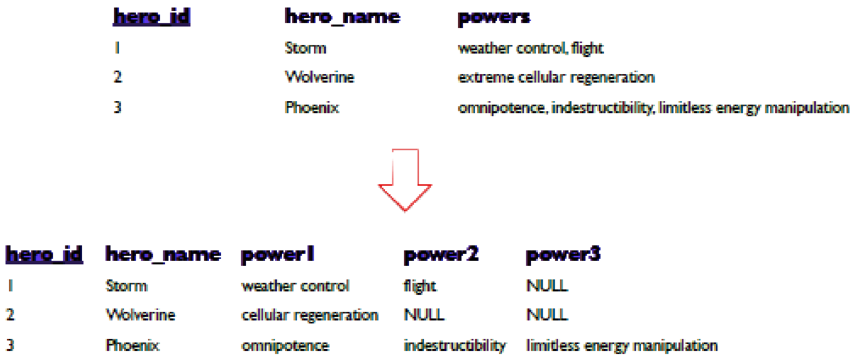
- b) **Replicating** the tuple for each multi-value
  - uses old key and multi-attribute as composite key



hero_id	hero_name	powers
1	Storm	weather control, flight
2	Wolverine	extreme cellular regeneration
3	Phoenix	omnipotence, indestructibility, limitless energy manipulation

hero_id	hero_name	powers
1	Storm	weather control
1	Storm	flight
2	Wolverine	extreme cellular regeneration
3	Phoenix	omnipotence
3	Phoenix	indestructibility
3	Phoenix	limitless energy manipulation

- c) Introducing an **own attribute** for each multi-value



hero_id	hero_name	powers
1	Storm	weather control, flight
2	Wolverine	extreme cellular regeneration
3	Phoenix	omnipotence, indestructibility, limitless energy manipulation

hero_id	hero_name	power1	power2	power3
1	Storm	weather control	flight	NULL
2	Wolverine	cellular regeneration	NULL	NULL
3	Phoenix	omnipotence	indestructibility	limitless energy manipulation

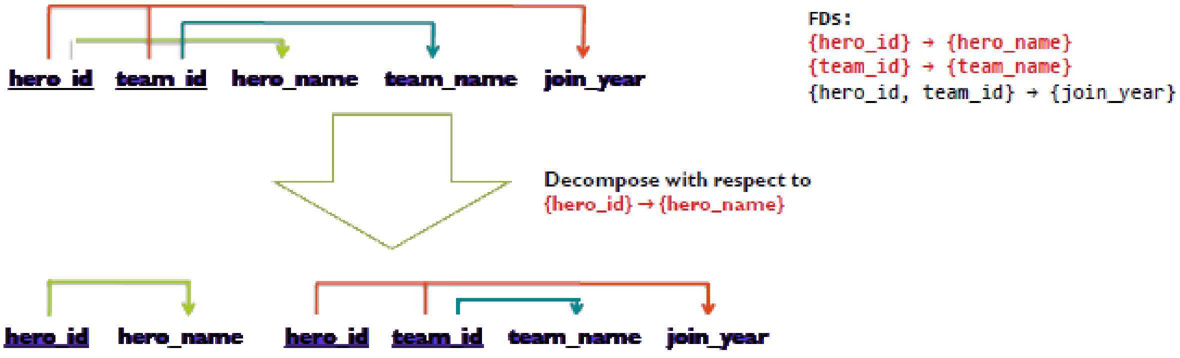
(Foreign key inside composite key)

2NF implied relation from part of the composite key

- **The second normal form (2NF)**
  - the 2NF aims to avoid attributes that are functionally dependent on proper subsets of keys
  - remember
    - a set of attributes X is a **(candidate) key** if and only if $X \rightarrow \{A_1, ..., A_n\}$ is a valid FD
    - an attribute A_i is a **key (or prime) attribute** if and only if it is contained in some key; otherwise, it is a **non-key (-prime) attribute**
  - **definition (2NF):**
    A relation schema is in **2NF** (wrt. a set of FDs) iff ...
    - it is in 1NF and
    - **no non-key attribute is functionally dependent on a proper subset of any candidate key.**

- Functional dependence on key parts is only a problem in relation schemas with composite keys
  - a (candidate) key is called **composite key** if it consists of more than one attribute
- **Corollary:**
  Every 1NF-relation without constant attributes and without **composite keys** is in 2NF.
  - 2NF is violated, if there is a **composite key** and some **non-key attribute** depends only on a **proper subset** of this composite key

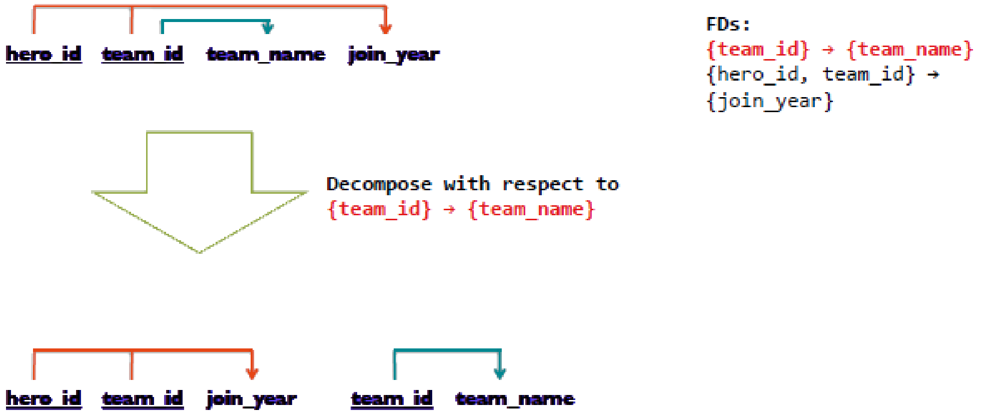
- Normalization into 2NF is achieved by **decomposition** according to the *non-2NF* FDs
  - if $X \rightarrow Y$ is a valid FD and X is a proper subset of some key, then decompose into $\alpha_1 = X \cup Y$ and $\alpha_2 = \{A_1, ..., A_n\} \setminus Y$
  - according to Heath's Theorem, this decomposition is **lossless**



FDs:
{hero_id} → {hero_name}
{team_id} → {team_name}
{hero_id, team_id} → {join_year}

Decompose with respect to
{hero_id} → {hero_name}

- Repeat this decomposition step for every created relation schema that is still not in 2NF



- Practical Implication of 2NF:
  - Normalized tables tend to focus on a single topic
    - Other topics are usually pulled in own tables
    - Some topic mixes remain

3NF Keep only foreign key

- The third normal form (3NF)
  - **Most relevant and practical normal form!**
  - A relation schema is in 3NF if and only if:
    - it is 2NF and
    - all non-key attribute are determined ONLY by the candidate key.

hero_id	hero_name	home_city_id	home_city_name
I1	Professor X	563	New York
I2	Wolverine	782	Alberta
I3	Cyclops	112	Anchorage
I4	Phoenix	563	New York

{hero_id} → {hero_name}
{hero_id} → {home_city_id}
{home_city_id} → {home_city_name}

Not in 3NF

  - the 3NF relies on the concept of **transitive FDs**
    - **Definition transitive FDs:**
      Given a set of FDs F, an FD $X \rightarrow Z \in F^+$ is **transitive** in F, if and only if there is an attribute set Y such that:
      - $X \rightarrow Y \in F^+$,
      - $Y \rightarrow X \notin F^+$, and
      - $Y \rightarrow Z \in F^+$.
    - No non-key attribute is transitively dependent on a key attribute
  - **Example**
    - {hero_id} → {hero_name}
    - {hero_id} → {home_city_id}
    - {hero_id} → {home_city_name}
    - {home_city_id} → {home_city_name}

hero_id	hero_name	home_city_id	home_city_name
I1	Professor X	563	New York
I2	Wolverine	782	Alberta
I3	Cyclops	112	Anchorage
I4	Phoenix	563	New York

Chapter 14.

The implicit goals of the design activity are information preservation and minimum redundancy.

Information preservation in terms of maintaining all concepts, including attribute types, entity types, and relationship types as well as generalization/specialization relationships, which are described using a model such as the EER model. Thus, the relational design must preserve all of these concepts, which are originally captured in the conceptual design after the conceptual to logical design mapping. Minimizing redundancy implies minimizing redundant storage of the same information and reducing the need for multiple updates to maintain consistency across multiple copies of the same information in response to real-world events that require making an update.

Successive normal forms are defined to meet a set of desirable constraints expressed using primary keys and functional dependencies. The normalization procedure consists of applying a series of tests to relations to meet these increasingly stringent requirements and decompose the relations when necessary.

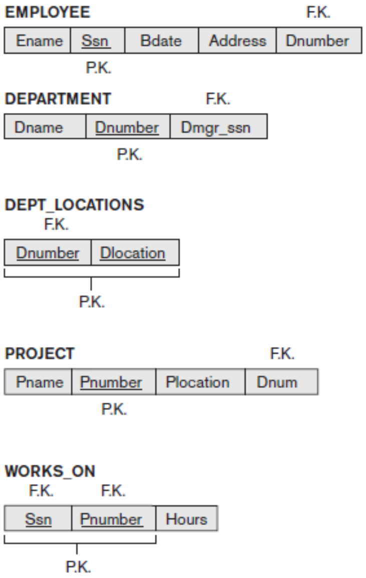
14.1 Informal Design Guidelines for Relation Schemas

Before discussing the formal theory of relational database design, we discuss four *informal guidelines* that may be used as *measures to determine the quality* of relation schema design:

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information in tuples
- Reducing the NULL values in tuples
- Disallowing the possibility of generating spurious tuples

These measures are not always independent of one another.

Figure 14.1
A simplified COMPANY relational database schema.



Guideline 1. Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation. Intuitively, if a relation schema corresponds to one entity type or one relationship type, it is straightforward to explain its meaning. Otherwise, if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

Guideline 2. Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.

Guideline 3. As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL. If NULLs are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation.

Guideline 4. Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated. Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

14.2 Functional Dependencies

Any relational database schema can be expressed as a single **universal** Relation Schema R with n Attributes $R = \{A_1, A_2, \ldots, A_n\}$. Such table will have a lot of rows and a lot of null values.

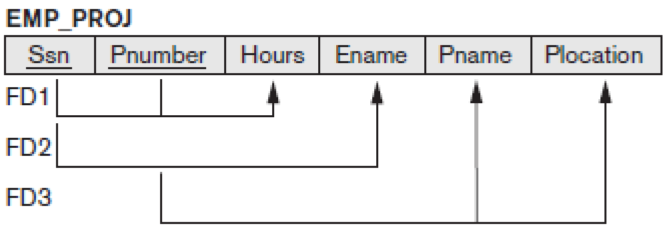
In that huge table, we can ~~identify~~ ask the owner for patterns, called “functional dependencies”.

That is attribute X implies attribute Y. X->Y. This combination of attributes are a proper subset of R. They also specify a constraint on the possible tuples that can be accepted in a valid state for R.

The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$. That is also called Y is functionally dependent on X. X is called teh left-hand side of the FD and y the right That also means that X is a candidate key of R. Then X → R.
Relation extensions r(R) that satisfy the functional dependency constraints are called legal relation states (or legal extensions) of R. Hence, the main use of functional dependencies is to describe further a relation schema R by specifying constraints on its attributes that must hold at all times.

A functional dependency is a property of the relation schema R, not of a particular legal relation state r of R. Therefore, an FD cannot be inferred automatically from a given relation extension r but must be defined explicitly by someone who knows the semantics of the attributes of R.

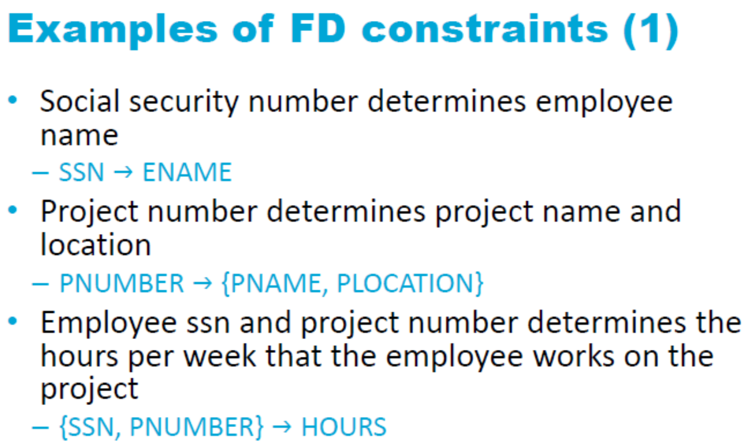
We can hower use counter examples to prove that certain FD do not hold.



These FD indicate that Hours and Ename are attributes of an entity that has ssn as primary key, and Pname and Plocation are attributes of an entity that has Pnumber has primary key.

This table violates 3NF, only attributes of the primary key should be in a table. Attributes of the foreign key should be kept at its own table.

Lecture notation for FD constraints



Examples of FD constraints (1)

- Social security number determines employee name
  - SSN → ENAME
- Project number determines project name and location
  - PNUMBER → {PNAME, PLOCATION}
- Employee ssn and project number determines the hours per week that the employee works on the project
  - {SSN, PNUMBER} → HOURS

BTW a relation is a table that explicty has foreign keys. A (composite) “key” can be redundant (scrapping things will keep it unique), a superkey is atomic (scrapping thigns will not keep it unique). Single value keys are superkeys by default (what else is left to remove? ;P)

14.3 Normal Forms Based on Primary keys

Prime attributes are attributes belonging to a candidate key.

Most practical relational design projects take one of the following two approaches:

- Perform a conceptual schema design using a conceptual model such as ER or EER and map the conceptual design into a set of relations.
- Design the relations based on external knowledge derived from an existing implementation of files or forms or reports.

Stack Overflow:

1NF

1NF is the most basic of normal forms - each cell in a table must contain only one piece of information, and there can be no duplicate rows.

2NF and 3NF are all about being dependent on the primary key. Recall that a primary key can be made up of multiple columns. As Chris said in his response:

The data depends on the key [1NF], the whole key [2NF] and nothing but the key [3NF] (so help me Codd).

2NF

Say you have a table containing courses that are taken in a certain semester, and you have the following data:

```
|-----Primary Key----|               uh oh |
                                           V
CourseID | SemesterID | #Places  | Course Name  |
------------------------------------------------|
IT101    |   2009-1   | 100      | Programming  |
IT101    |   2009-2   | 100      | Programming  |
IT102    |   2009-1   | 200      | Databases    |
IT102    |   2010-1   | 150      | Databases    |
IT103    |   2009-2   | 120      | Web Design   |
```

This is **not in 2NF**, because the fourth column does not rely upon the *entire* key - but only a part of it. The course name is dependent on the Course's ID, but has nothing to do with which semester it's taken in. Thus, as you can see, we have duplicate information - several rows telling us that IT101 is programming, and IT102 is Databases. So we fix that by moving the course name into another table, where CourseID is the ENTIRE key.

```
Primary Key |

CourseID    |  Course Name |
---------------------------|
IT101       | Programming  |
IT102       | Databases    |
IT103       | Web Design   |
```

No redundancy!

3NF

Okay, so let's say we also add the name of the teacher of the course, and some details about them, into the RDBMS:

```
|-----Primary Key----|                          uh oh |
                                                     V
Course  |  Semester  |  #Places   |  TeacherID  | TeacherName  |
---------------------------------------------------------------|
IT101   |   2009-1   |  100       |  332        |  Mr Jones    |
IT101   |   2009-2   |  100       |  332        |  Mr Jones    |
IT102   |   2009-1   |  200       |  495        |  Mr Bentley  |
IT102   |   2010-1   |  150       |  332        |  Mr Jones    |
IT103   |   2009-2   |  120       |  242        |  Mrs Smith   |
```

Now hopefully it should be obvious that TeacherName is dependent on TeacherID - so this is **not in 3NF**. To fix this, we do much the same as we did in 2NF - take the TeacherName field out of this table, and put it in its own, which has TeacherID as the key.

```
 Primary Key |

 TeacherID   | TeacherName  |
 ---------------------------|
 332         |  Mr Jones    |
 495         |  Mr Bentley  |
 242         |  Mrs Smith   |
```

No redundancy!!

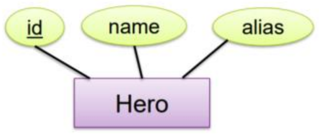
One important thing to remember is that if something is not in 1NF, it is not in 2NF or 3NF either. So each additional Normal Form requires *everything* that the lower normal forms had, plus some extra conditions, which must *all* be fulfilled.

Lecture 6

Conceptual to Logical

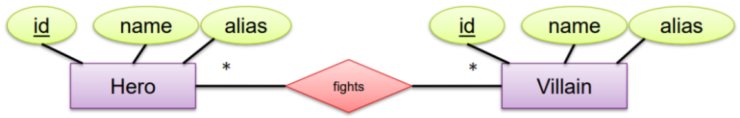
- Conceptual Schema (E)ER
  - „Which entities should be stored, what are their properties, and how are they related?"
- (Relational) Logical Schema $R_1(\underline{attr}, \underline{attr} \to R_2(), \ldots)$
  - "Which relations should exist, which attributes do they have, what are the domains (data types) of the attributes, and constraints should hold?"
- Physical Schema SQL
  - "Where and how to store relations, what to index, what meta-data / statistics to collect, etc.?"

- Converting a simple Entity Type into a relation schema:



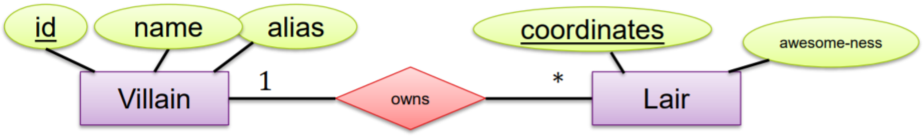
Hero($\underline{\text{id}}$, name, alias)

- Converting an n:m relationship type into a relation schema:
  - Relationship type becomes a separate relation schema
    - Links entities of the respective types by using their foreign keys



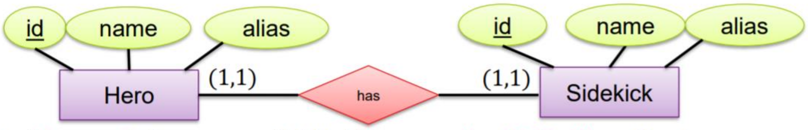
Hero($\underline{\text{id}}$, name, alias)
hero_fights_villain($\underline{\text{hero}}$ → **Hero**, $\underline{\text{villain}}$→**Villain**)
Villain(*$\underline{\text{id}}$, name, alias*)

- Converting an 1:m relationship type a relation schema:
  - Entity Type at 1-side can only participate once at the relationship type
    - => Push relationship type to the 1-side



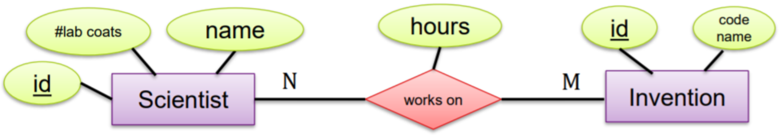
Villain(id, name, alias)
Lair(coordinates, awesomeness, **owner→Villain**)

- Converting a 1:1 relationship type a relation schema:
  - A little bit tricky...
    - Cannot be expressed just by the relation schemas...
    - Just choose one side as the 1-side and implement it just like a 1:m relationship type



Hero(id, name, alias) **Sidekick**(id, name, alias, **bigBuddy** → **Hero**)

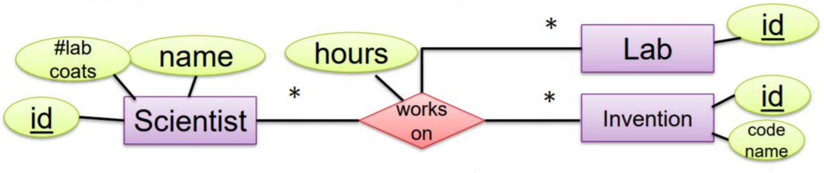
- How to deal with attributes attached to a relationship type
  - For, n:m, just put



Scientist(id, name, #lab_coats) *Invention(id, code_name)*

scientist_works_on_invention(
scientist → Scientist,
invention → Invention,
hours
)

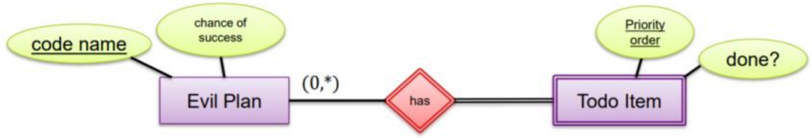
- What about n-ary relationship types? (n>2)
  - Just apply the exact same approaches:



sc_wrk_on_inv_in_lab(
scientist → Scientist,
invention → Invention,
lab → Lab
hours)

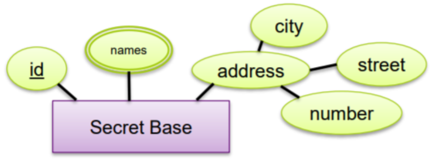
- Converting a weak entity into a relation schema:
  - Weak entities are only unique together with the entity at the **identifying relationship**

  => Follow identifying relationship and inherit respective foreign keys



Evil_Plan(*code_name*, *chance_of_success*)
todo_item(priority_order, evil_plan → Evil_Plan, done)

- How to deal with multi-attributes and composite attributes
  - **composition:** just flatten it
  - **multi-attribute:** treat it like a weak entity



Version 1:
Secret_Base(id, addr_city, addr_street, addr_number)
base_name (hideout → Secret_Base, name)

(own relation for names, adress attributes flattened into Secet_Base)

Version 1:
Secret_Base(id, addr_city, addr_street, addr_number)
base_name (hideout → Secret_Base, name)

(own relation for names, adress attributes flattened into Secet_Base)

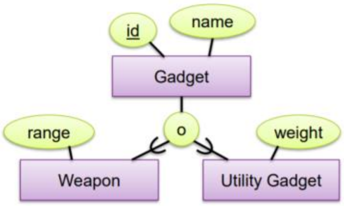
Version 2:
Secret_Base(id)
base_name (hideout → Secret_Base, name)
adresses(city, street, number, base → Secret_Base)

(own relation for names and address; this basically represents now a 1:m relationship between bases and addresses so likely not as good as version 1?)

Version 3:
Secret_Base(id, addr_city, addr_street, addr_number, name1, name2, name3, name4, name5, name6, name7)

(mhh….☹ Maybe. Depends. In most cases this is probably a very bad solution….- could even argue that it's wrong because what happens if something has 8 names? Still people do this.)

- Converting types with inherited attributes/ relations into a relation schema:
  - Can be implemented in many ways
    - more than 3…



Gadget(id, name)
Weapon(gadget → Gadget, range)
Util_gadget(gadget → Gadget, weight)

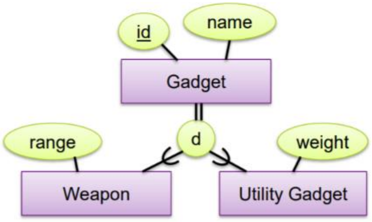
Gadget(id, name)
Weapon(id, name, range)
Utility(id, name, weight)

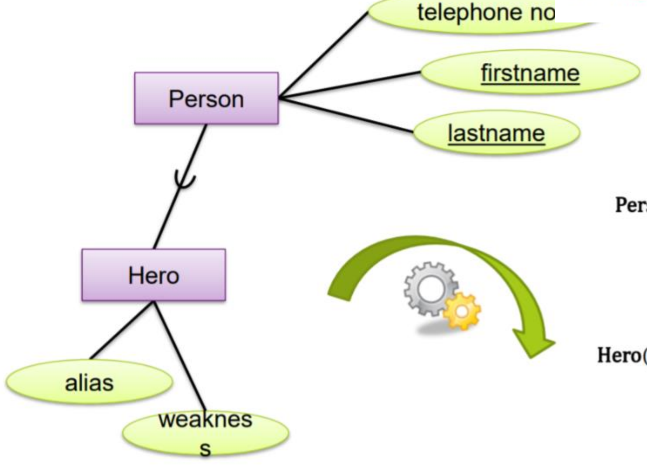
- Version 3:

Gadget(id, name, range, weight)

Inheritance Version 2:
(now total disjoint)

Weapon(id name, range)
Utility(id, name, weight)





Person(firstname: string,
lastname: string,
telephone_no: string)

Hero(firstname: string,
lastname: string,
alias: string,
weakness: string,
(firstname, lastname) → Person
)

Week 4. SQL

Lecture 7 – SQL

SQL is a high-level declarative language, that means the user defines what result he expects and know how it is executed. The exeuction of the relational algebra is left to the optimizer by the DBMS.

SQL is a comprehensive database language: It has statements for data definitions, queries, and updates. Hence, it is both a DDL (**data definition language**, used by the DBA and by database designers to define conceptual and internal schemas.) *and* a DML (**data manipulation language** i.e. SCRUD). This course uses postgreSQL https://www.postgresql.org/docs/11/index.html

6.1 SQL Data Definition and Data Types

SQL uses the terms table, row, and column for the formal relational model terms relation, tuple, and attribute, respectively.

SQL command for data definition is the CREATE statement, which can be used to create databases (schemas), tables (relations), types, and domains, as well as other constructs such as views, assertions, and triggers.

An SQL schema (database) is identified by a schema name and includes an authorization identifier to indicate the user or account who owns the schema, as well as descriptors for each element in the schema. Schema elements include tables, types, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema.

A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions. Although these can be changed later.

the following statement creates a schema called COMPANY owned by the user with authorization identifier 'Jsmith'.

CREATE SCHEMA COMPANY **AUTHORIZATION** 'Jsmith';

The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

A catalog is a collection of schemas. A catalog always contains a special schema called INFORMATION_SCHEMA, which provides information on all the schemas in the catalog and all the element descriptors in these schemas. Integrity constraints such as referential integrity can be defined between relations only if they exist in schemas within the same catalog. Schemas within the same catalog can also share certain elements, such as type and domain definitions.

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using ALTER TABLE.

Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing:

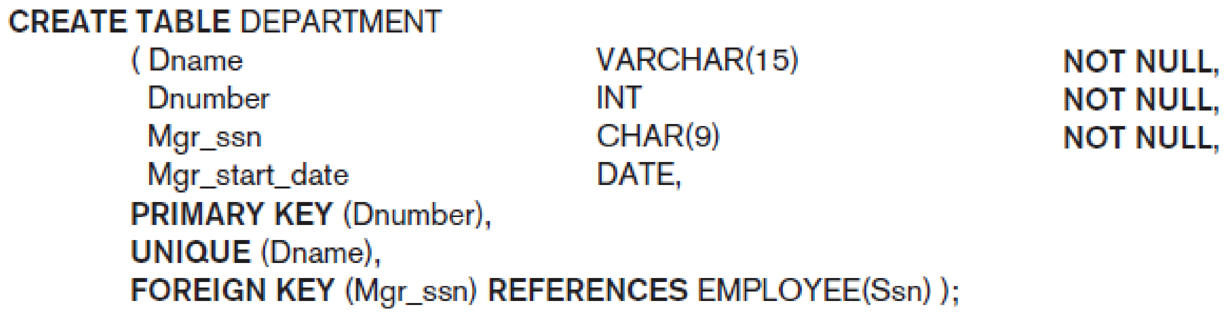
CREATE TABLE COMPANY.EMPLOYEE

rather than

CREATE TABLE EMPLOYEE

In PostgreSQL we are always connected to a particular database/schema. So appending is not needed.

Example of table creation with constraints and keys:



```
CREATE TABLE DEPARTMENT
        ( Dname                 VARCHAR(15)             NOT NULL,
          Dnumber               INT                     NOT NULL,
          Mgr_ssn               CHAR(9)                 NOT NULL,
          Mgr_start_date        DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
```

The relations declared through CREATE TABLE statements are called base tables (or base relations); this means that the table and its rows are actually created and stored as a file by the DBMS. Base relations are distinguished from virtual relations, created through the CREATE VIEW statement which do not correspond to a phyisical phile but a query. (It may correspond to a physical phile that contains the query script).

Declaring dependencies before the other tables have been created can lead to errors. Therefore it is often prefered to create all the tables without keys and constraints and later alter them to apply the dependencies and constraints.

Data types

The basic data types available for attributes include numeric, character string, bit string, Boolean, date, and time. For numeric the most common are INT and DOUBLE

When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is *case sensitive*. String values are stored in n/char(length) or n/varchar(length), the first being of a fixed length and the second specifying the max length.

Difference between nvarchar and varchar (also for nchar and char): The n prefix stands for "national character set". An nvarchar column can store multi-byte characters such as Unicode data. A varchar column is restricted to an 8-bit codepage. Some people think that varchar should be used because it takes up less space. I believe this is not the correct answer. Codepage incompatabilities are a pain, and Unicode is the cure for codepage problems. With cheap disk and memory nowadays, there is really no reason to waste time mucking around with code pages anymore [source].

For example, if the value ‘Smith’ is for an attribute of type CHAR(10), it is padded with five blank characters to become ‘Smith’ if needed. Padded blanks are generally ignored when strings are compared. For comparison purposes, strings are considered ordered in alphabetic (or lexicographic) order; if a string str1 appears before another string str2 in alphabetic order, then str1 is considered to be less than str2. There is also a concatenation operator denoted by || (double vertical bar) that can concatenate two strings.

A Boolean data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.

Bit-string data types are either of fixed length n—BIT(n)—or varying length— BIT VARYING(n), where n is the maximum number of bits. Another variable-length bitstring data type called BINARY LARGE OBJECT or BLOB is also available to specify columns that have large binary values, such as images. As for CLOB, the maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G). For example, BLOB(30G) specifies a maximum length of 30 gigabits.

The DATE data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS. Only valid dates and times should be allowed by the SQL implementation. This implies that months should be between 1 and 12 and days must be between 01 and 31; furthermore, a day should be a valid day for the corresponding month. The < (less than) comparison can be used with dates or times—an earlier date is considered to be smaller than a later date, and similarly with time. Literal values are represented by single-quoted strings preceded by the keyword DATE or TIME; for example, DATE '2014-09-27' or TIME '09:12:47'.

A timestamp data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier.

Another data type related to DATE, TIME, and TIMESTAMP is the INTERVAL data type. This specifies an interval—a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

The format of DATE, TIME, and TIMESTAMP can be considered as a special type of string. Hence, they can generally be used in string comparisons by being cast (or coerced or converted) into the equivalent strings.

It is possible to specify the data type of each attribute directly, alternatively, a domain can be declared, and the domain name can be used with the attribute specification. This makes it easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement:

CREATE DOMAIN SSN_TYPE **AS** CHAR(9);

Constraints

If a primary key has a single attribute, the clause can follow the attribute directly. For example:

KvkNummer INT **PRIMARY KEY**,

Same goes for UNIQUE, NOT NULL (which is already implied for primary keys) DEFAULT *value*

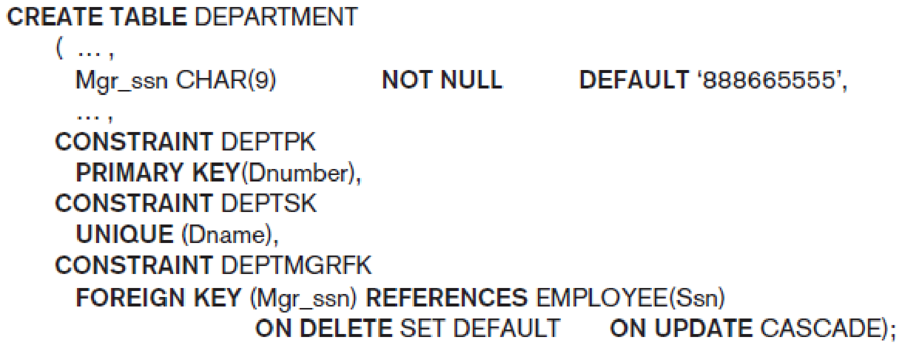
To declare a composite primary key:

PRIMARY KEY(coulmn1, column2, …)

Same goes for composite (and single) FOREIGN KEY.

Referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is updated. The default action that SQL takes for an integrity violation is to reject the update operation that will cause a violation, which is known as the RESTRICT option.

However, the schema designer can specify an alternative action to be taken by attaching a referential triggered action clause to any foreign key constraint. The options include SET NULL, CASCADE, and SET DEFAULT. An example with multiple constraints below:



```
CREATE TABLE DEPARTMENT
    ( … ,
      Mgr_ssn CHAR(9)          NOT NULL        DEFAULT '888665555',
      … ,
    CONSTRAINT DEPTPK
      PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
      UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
      FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                      ON DELETE SET DEFAULT      ON UPDATE CASCADE);
```

You can see that a constraint may be given a constraint name, following the keyword CONSTRAINT. The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint. Giving names to constraints is optional. It is also possible to temporarily defer a constraint until the end of a transaction.

CHECK (Column > 0 **AND** Column < 21); or CHECK (StartDate < EndDate); so you check for a boolean expression that returns 1 or 0. In that sense, it is possible to create a user-defined function that returns 1 or 0 by passing parameters from different tables as input. Here's an example check constraint using a function:

```
ALTER TABLE YourTable

ADD CONSTRAINT chk_CheckFunction

CHECK (dbo.CheckFunction() = 1)
```

Where you can define the function like:

```
CREATE FUNCTION dbo.CheckFunction()

RETURNS INT

AS BEGIN

  RETURN (SELECT 1 /* Query with different table inputs that either returns 1 or 0*/)

END
```

6.3 Basic Retrieval Queries in SQL

The basic form of the SELECT statement, sometimes called a mapping or a select-from-where block, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

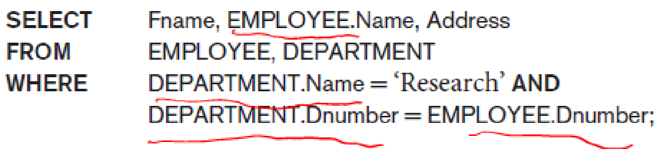
SELECT <attribute list>
FROM <table list>
WHERE <condition>;

By the way != in SQL is <>, && is AND || is OR (because || in SQL is the concatenate operator).

SQL will iterate through the whole set of tuples and evaluate them against the boolean condition

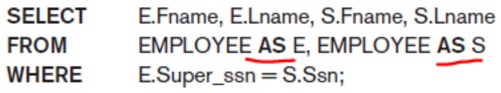
A query that involves only selection and join conditions plus projection attributes is known as a select-project-join query. The next example is a select-project-join query with two join conditions.

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in different tables. If this is the case, and a multitable query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity. This is done by prefixing the relation name to the attribute name and separating the two by a period.



```
SELECT   Fname, EMPLOYEE.Name, Address
FROM     EMPLOYEE, DEPARTMENT
WHERE    DEPARTMENT.Name = 'Research' AND
         DEPARTMENT.Dnumber = EMPLOYEE.Dnumber;
```

The ambiguity of attribute names also arises in the case of queries that refer to the same table twice, as in the following example. But we can use AS newName to define the new name of a table and a column as well.



```
SELECT   E.Fname, E.Lname, S.Fname, S.Lname
FROM     EMPLOYEE AS E, EMPLOYEE AS S
WHERE    E.Super_ssn = S.Ssn;
```

If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—all possible tuple combinations—of these relations is selected. This will often result in incorrect and very large relations.

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an asterisk (*), which stands for all the attributes. The * can also be prefixed by the relation name or alias; for example, EMPLOYEE.* refers to all attributes of the EMPLOYEE table.

Duplicates

As we mentioned earlier, SQL usually treats a table not as a set but rather as a multiset; duplicate tuples can appear more than once in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

■ Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.

■ The user may want to see duplicate tuples in the result of a query.

■ When an aggregate function is applied to tuples, in most cases we do not want to eliminate duplicates.

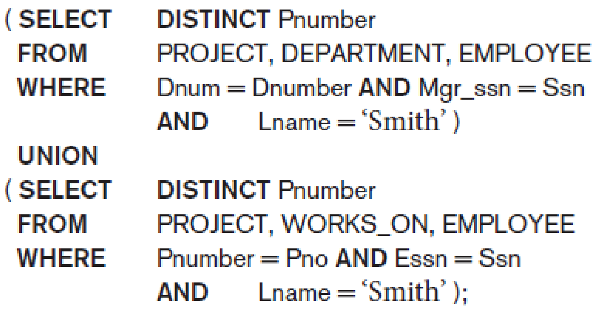
Duplicates are not possible if you have a primary key.

Anyway the DISTINCT keyword after SELECT will display unique tuples.

SELECT ALL is already implied when only writing SELECT

Set Operators

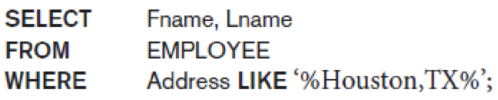
It is possible to use the set operations from mathematical set theory, set union (UNION), set difference (EXCEPT), and set intersection (INTERSECT). These set operations apply only to typecompatible relations, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations.



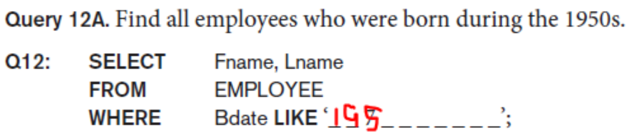
```
( SELECT   DISTINCT Pnumber
  FROM     PROJECT, DEPARTMENT, EMPLOYEE
  WHERE    Dnum = Dnumber AND Mgr_ssn = Ssn
           AND     Lname = 'Smith' )
  UNION
( SELECT   DISTINCT Pnumber
  FROM     PROJECT, WORKS_ON, EMPLOYEE
  WHERE    Pnumber = Pno AND Essn = Ssn
           AND     Lname = 'Smith' );
```

Substring Pattern Matching

The first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. This can be used for string pattern matching. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character.

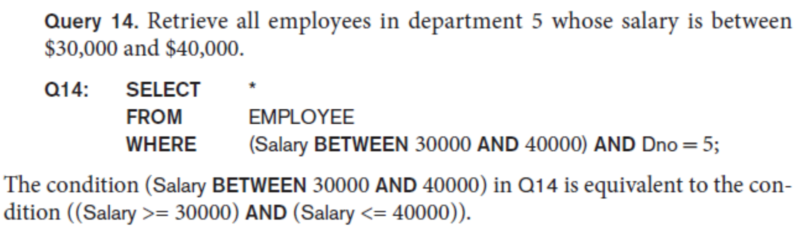


```
SELECT   Fname, Lname
FROM     EMPLOYEE
WHERE    Address LIKE '%Houston,TX%';
```



Query 12A. Find all employees who were born during the 1950s.

```
Q12:   SELECT   Fname, Lname
       FROM     EMPLOYEE
       WHERE    Bdate LIKE '195_______';
```



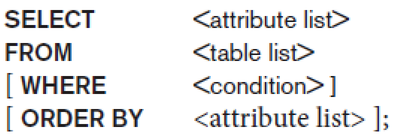
Query 14. Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
Q14:   SELECT    *
       FROM      EMPLOYEE
       WHERE     (Salary BETWEEN 30000 AND 40000) AND Dno = 5;
```

The condition (Salary **BETWEEN** 30000 **AND** 40000) in Q14 is equivalent to the condition ((Salary >= 30000) **AND** (Salary <= 40000)).

Ordering

The default order is in ascending order of values. We can specify the keyword DESC if we want to see the result in a descending order of values. The keyword ASC can be used to specify ascending order explicitly. Order by goes after where.

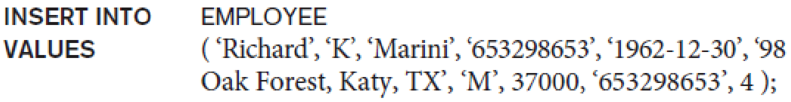


```
SELECT        <attribute list>
FROM          <table list>
[ WHERE       <condition> ]
[ ORDER BY    <attribute list> ];
```

INSERT, DELETE and UPDATE

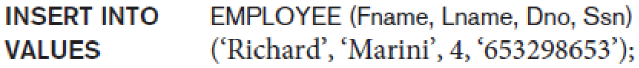
In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE.

In its simplest form, INSERT is used to add a single tuple (row) to a relation (table). We must specify the relation name and a list of values for the tuple. The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command



```
INSERT INTO    EMPLOYEE
VALUES         ( 'Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
               Oak Forest, Katy, TX', 'M', 37000, '653298653', 4 );
```

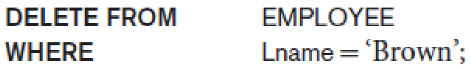
If you don't know the order or if you only want to add partial elements then you can explicitly list these attributes and then it's values, in the same order.



```
INSERT INTO    EMPLOYEE (Fname, Lname, Dno, Ssn)
VALUES         ('Richard', 'Marini', 4, '653298653');
```

Just be sure that the missing attributes don't have NOT NULL constraint. It is also possible to insert into a relation multiple tuples separated by commas in a single INSERT command. The attribute values forming each tuple are enclosed in parentheses.

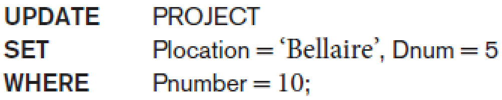
The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if referential triggered actions are specified in the referential integrity constraints of the DDL. Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table. We must use the DROP TABLE command to remove the table definition. DROP TABLE EMPLOYEE;



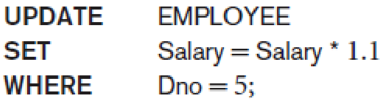
```
DELETE FROM    EMPLOYEE
WHERE          Lname = 'Brown';
```

```
DELETE FROM    EMPLOYEE;
```

The UPDATE command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints of the DDL. An additional SET clause in the UPDATE command specifies the attributes to be modified and their new values:



```
UPDATE    PROJECT
SET       Plocation = 'Bellaire', Dnum = 5
WHERE     Pnumber = 10;
```



```
UPDATE    EMPLOYEE
SET       Salary = Salary * 1.1
WHERE     Dno = 5;
```

(10% raise to employee 5)

Notice that

Each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.

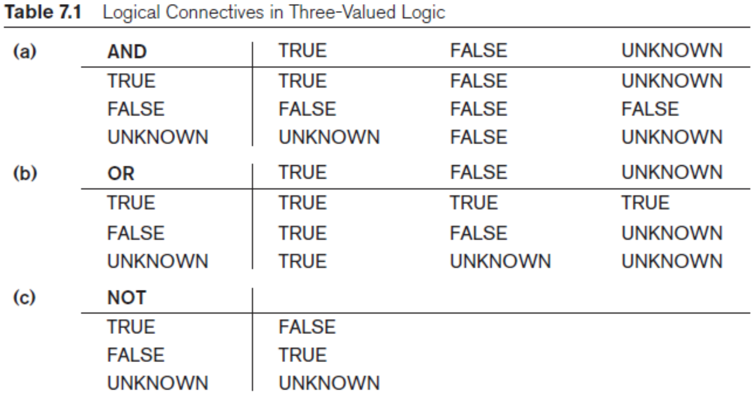
Comparison involving NULL and three-valued logic

SQL has various rules for dealing with NULL values. NULL is used to represent a missing value, but that it usually has one of three different interpretations—value unknown (value exists but is not known, or it is not known whether or not the value exists), value not available (value exists but is purposely withheld), or value not applicable (the attribute does not apply to this tuple or is undefined for this tuple).

It is often not possible to determine which of the meanings is intended; for example, a NULL for the home phone of a person can have any of the three meanings. Hence, SQL does not distinguish among the different meanings of NULL.

Each individual NULL value is considered to be different from every other NULL value in the various database records. When a record with NULL in one of its attributes is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE.

Table 7.1 Logical Connectives in Three-Valued Logic



(a)

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)

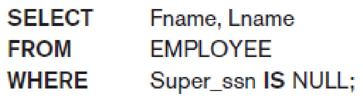
OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

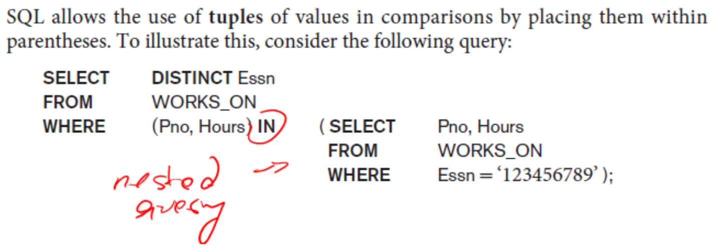
NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected (unless it is outer join).

SQL allows queries that check whether an attribute value is NULL. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators IS or IS NOT**. This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate.**



```
SELECT     Fname, Lname
FROM       EMPLOYEE
WHERE      Super_ssn IS NULL;
```



SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

```
SELECT   DISTINCT Essn
FROM     WORKS_ON
WHERE    (Pno, Hours) IN   ( SELECT   Pno, Hours
                             FROM     WORKS_ON
                             WHERE    Essn = ‘123456789’ );
```

nested query

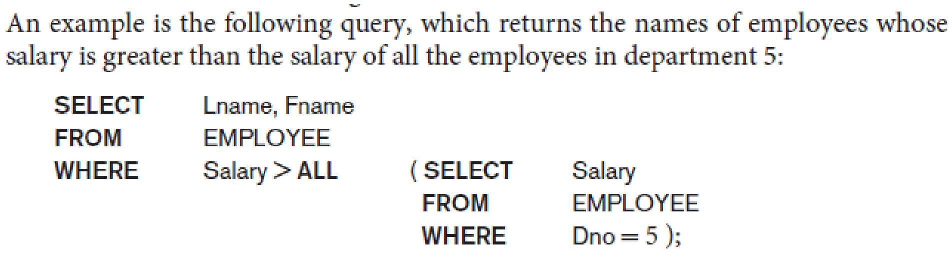
Nested Queries

IN can also refer to a list of values:

SELECT name, population **FROM** world **WHERE** name **IN** ('Brazil', 'Russia', 'India', 'China');

= ANY and = **SOME** have the same effect as **IN**

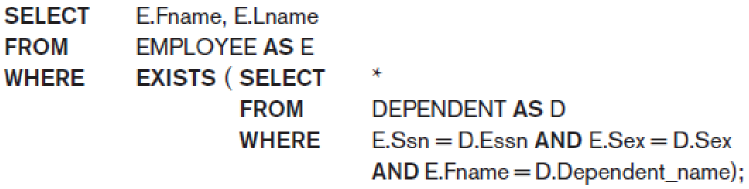
Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>. The keyword **ALL** can also be combined with each of these operators.



An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

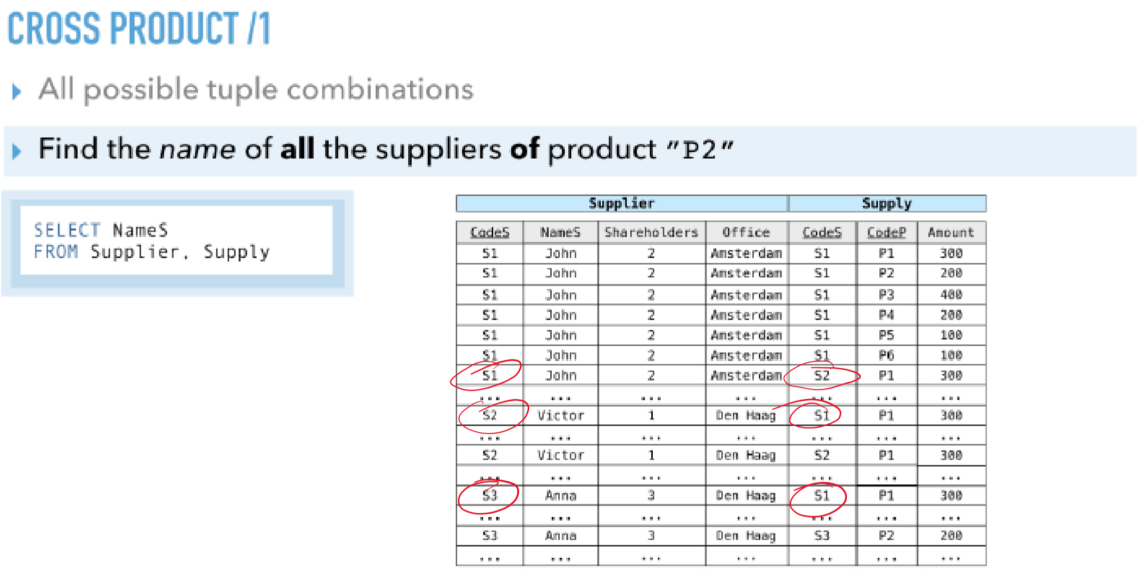
```
SELECT    Lname, Fname
FROM      EMPLOYEE
WHERE     Salary > ALL    ( SELECT    Salary
                            FROM      EMPLOYEE
                            WHERE     Dno = 5 );
```

EXISTS and NOT EXISTS in SQL is used to check whether the result of a nested query is *empty* (contains no tuples) or not.



```
SELECT    E.Fname, E.Lname
FROM      EMPLOYEE AS E
WHERE     EXISTS ( SELECT    *
                   FROM      DEPENDENT AS D
                   WHERE     E.Ssn = D.Essn AND E.Sex = D.Sex
                             AND E.Fname = D.Dependent_name);
```

Lecture - Joins



CROSS PRODUCT /1

- All possible tuple combinations
- Find the *name* of **all** the suppliers **of** product "P2"

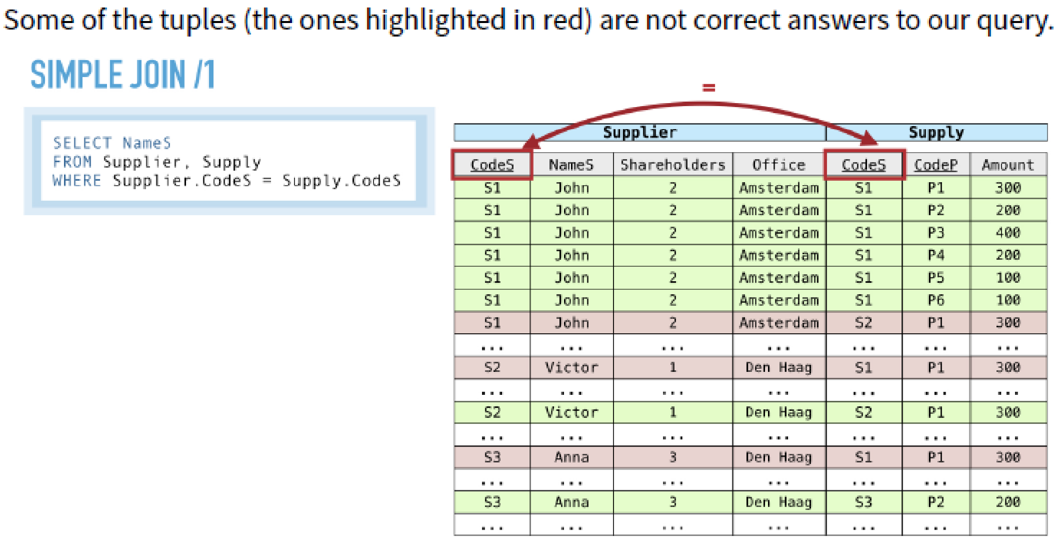
```
SELECT NameS
FROM Supplier, Supply
```

Supplier				Supply		
CodeS	NameS	Shareholders	Office	CodeS	CodeP	Amount
S1	John	2	Amsterdam	S1	P1	300
S1	John	2	Amsterdam	S1	P2	200
S1	John	2	Amsterdam	S1	P3	400
S1	John	2	Amsterdam	S1	P4	200
S1	John	2	Amsterdam	S1	P5	100
S1	John	2	Amsterdam	S1	P6	100
S1	John	2	Amsterdam	S2	P1	300
...	...	...	...	...	...	...
S2	Victor	1	Den Haag	S1	P1	300
...	...	...	...	...	...	...
S2	Victor	1	Den Haag	S2	P1	300
...	...	...	...	...	...	...
S3	Anna	3	Den Haag	S1	P1	300
...	...	...	...	...	...	...
S3	Anna	3	Den Haag	S3	P2	200
...	...	...	...	...	...	...

Figure 37: Slide 53 - Cross Product

When we need to formulate a query that involves rows belonging to **more than one table**, the argument of the FROM clause is given as a list of tables. The conditions in the WHERE clause are applied to the *cartesian product* (or **cross product**) of these tables. Recall that a Cartesian product of two sets A and B is the the set of all the possible pairs in which the first element belongs to A and the second to B. In the PostgreSQL documentation, the cartesian product is referred to as CROSS JOIN.

For instance, in the figure, we can see how tuples from the SUPPLIER table are combined with tuples from the SUPPLY table.



Some of the tuples (the ones highlighted in red) are not correct answers to our query.

SIMPLE JOIN /1

```
SELECT NameS
FROM Supplier, Supply
WHERE Supplier.CodeS = Supply.CodeS
```

Supplier				Supply		
CodeS	NameS	Shareholders	Office	CodeS	CodeP	Amount
S1	John	2	Amsterdam	S1	P1	300
S1	John	2	Amsterdam	S1	P2	200
S1	John	2	Amsterdam	S1	P3	400
S1	John	2	Amsterdam	S1	P4	200
S1	John	2	Amsterdam	S1	P5	100
S1	John	2	Amsterdam	S1	P6	100
S1	John	2	Amsterdam	S2	P1	300
...	...	...	...	...	...	...
S2	Victor	1	Den Haag	S1	P1	300
...	...	...	...	...	...	...
S2	Victor	1	Den Haag	S2	P1	300
...	...	...	...	...	...	...
S3	Anna	3	Den Haag	S1	P1	300
...	...	...	...	...	...	...
S3	Anna	3	Den Haag	S3	P2	200
...	...	...	...	...	...	...

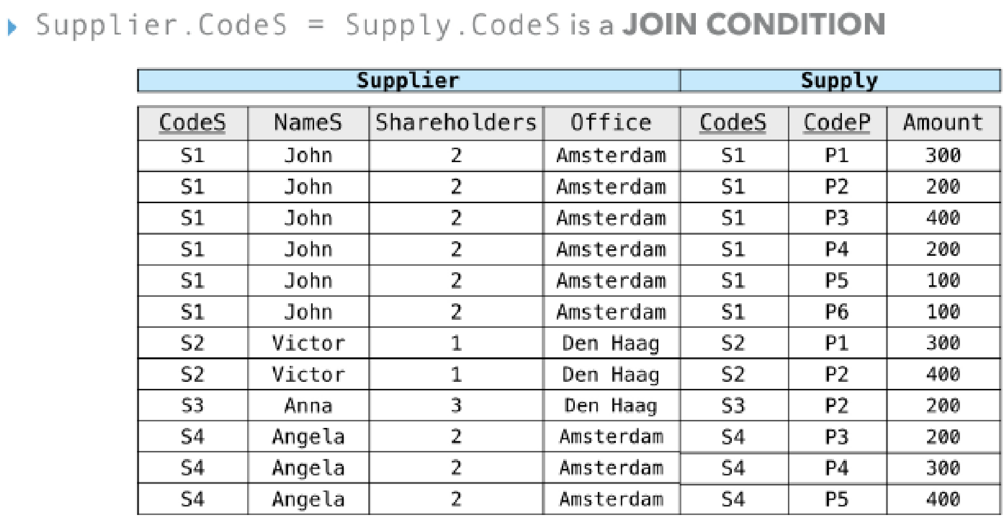
This is why a JOIN operation can be specified, by explicitly indicating comparisons between attributes of different tables.

The JOIN operator is arguably the most important one in relational algebra, and, in general, in databases. The JOIN allows us to establish connections among data contained in different relations, comparing the values contained in them and thus using the fundamental characteristics of the model, that of being **value-based**.

In the above query, we perform a JOIN by explicitly indicating a *"connection"* between attributes in the two tables, in the WHERE clause. This means that, after the cartesian product between the two tables is calculated, only those tuples having the same value for the CodeS attributes are kept.

Note the use of the *dot* operator to identify the tables from which attributes are extracted. For example, Supplier.CodeS denotes the CodeS attribute of the table Supplier. This use is common in many programming languages, to identify the fields of a structured variable.

It is necessary to use this notation when the tables listed in the FROM clause have attributes with the same name, in order to distinguish among the references to the homonym attributes.

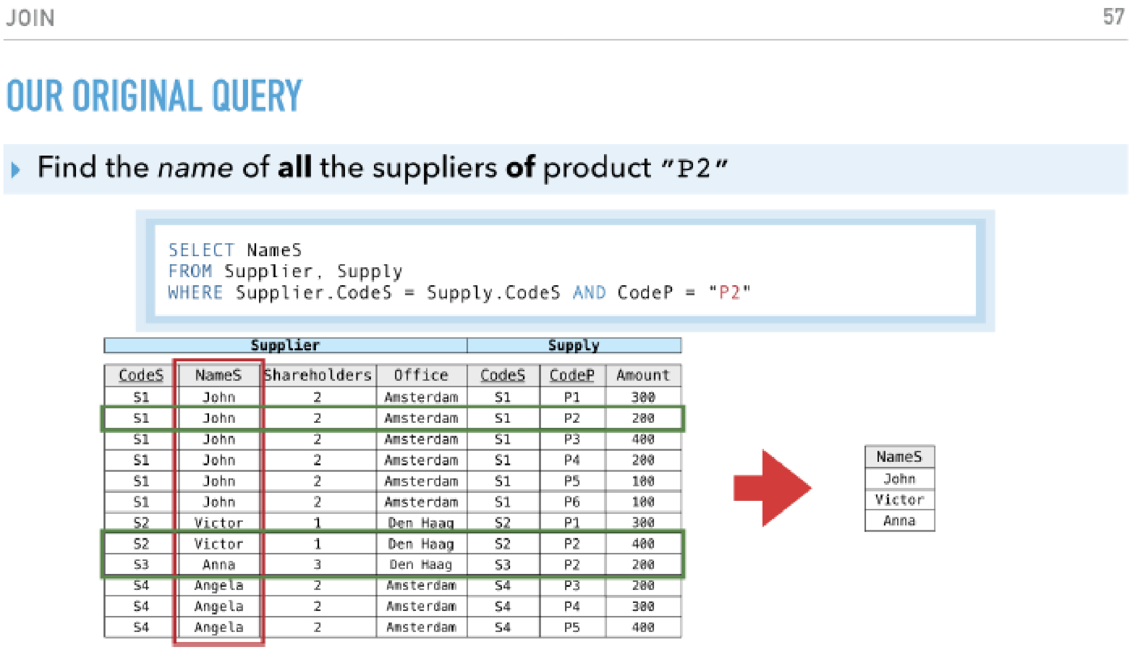


- Supplier.CodeS = Supply.CodeS is a **JOIN CONDITION**

Supplier				Supply		
CodeS	NameS	Shareholders	Office	CodeS	CodeP	Amount
S1	John	2	Amsterdam	S1	P1	300
S1	John	2	Amsterdam	S1	P2	200
S1	John	2	Amsterdam	S1	P3	400
S1	John	2	Amsterdam	S1	P4	200
S1	John	2	Amsterdam	S1	P5	100
S1	John	2	Amsterdam	S1	P6	100
S2	Victor	1	Den Haag	S2	P1	300
S2	Victor	1	Den Haag	S2	P2	400
S3	Anna	3	Den Haag	S3	P2	200
S4	Angela	2	Amsterdam	S4	P3	200
S4	Angela	2	Amsterdam	S4	P4	300
S4	Angela	2	Amsterdam	S4	P5	400

Figure 40: Slide 56 - Simple JOIN

The content of the WHERE clause is what is called a JOIN condition, and in the slide we can appreciate the result of its application to the result set.



JOIN 57

OUR ORIGINAL QUERY

- Find the *name* of **all** the suppliers **of** product "P2"

```
SELECT NameS
FROM Supplier, Supply
WHERE Supplier.CodeS = Supply.CodeS AND CodeP = "P2"
```

Supplier				Supply		
CodeS	NameS	Shareholders	Office	CodeS	CodeP	Amount
S1	John	2	Amsterdam	S1	P1	300
S1	John	2	Amsterdam	S1	P2	200
S1	John	2	Amsterdam	S1	P3	400
S1	John	2	Amsterdam	S1	P4	200
S1	John	2	Amsterdam	S1	P5	100
S1	John	2	Amsterdam	S1	P6	100
S2	Victor	1	Den Haag	S2	P1	300
S2	Victor	1	Den Haag	S2	P2	400
S3	Anna	3	Den Haag	S3	P2	200
S4	Angela	2	Amsterdam	S4	P3	200
S4	Angela	2	Amsterdam	S4	P4	300
S4	Angela	2	Amsterdam	S4	P5	400

NameS
John
Victor
Anna

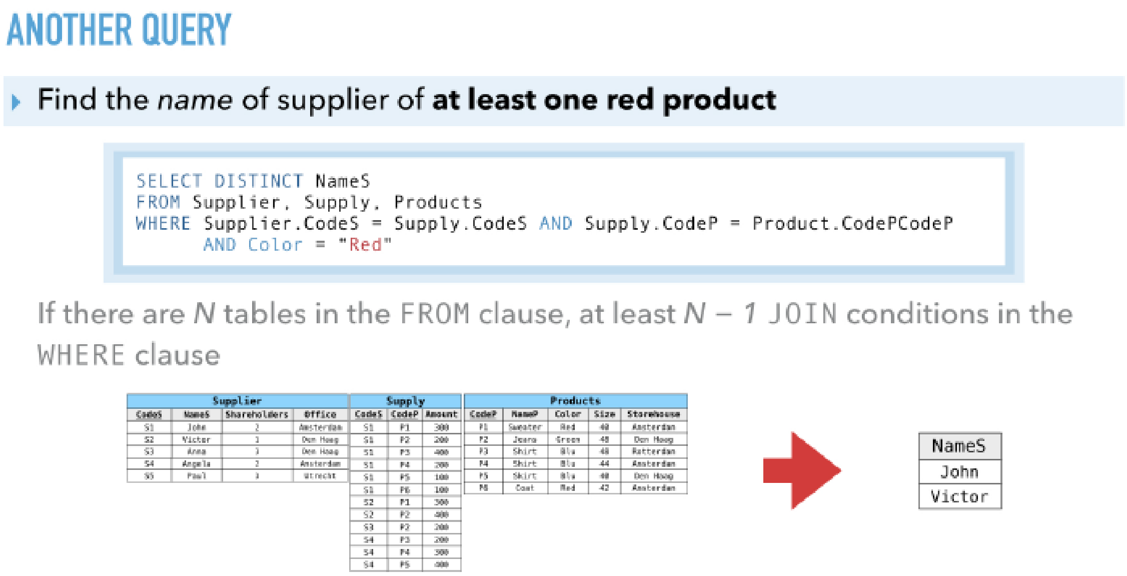
Figure 41: Slide 57 - Our original query

With this new powerful condition in our toolkit, we are now able to express more complex queries. So, in this examples, we can find *the name of all suppliers of a given product* P2, by joining the Supplier and Aupply table, and then add a new condition on the product CodeP.

8.6.1 Example Query in DB3

The query below retrieves the list of movies (title) having at least one cast member.

```
SELECT name
FROM person_100k, cast_info_100k
WHERE person_100k.id = cast_info_100k.person_id
```



ANOTHER QUERY

- Find the *name* of supplier of **at least one red product**

```
SELECT DISTINCT NameS
FROM Supplier, Supply, Products
WHERE Supplier.CodeS = Supply.CodeS AND Supply.CodeP = Product.CodePCodeP
      AND Color = "Red"
```

If there are *N* tables in the FROM clause, at least *N − 1* JOIN conditions in the WHERE clause

Supplier			
CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supply		
CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

Products				
CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam

NameS
John
Victor

Figure 42: Slide 58 - Another Query

The FROM clause can accommodate more than 2 tables.

In this case, there WHERE clause should specify *N-1* JOIN conditions, to allow the join of *N* tables.

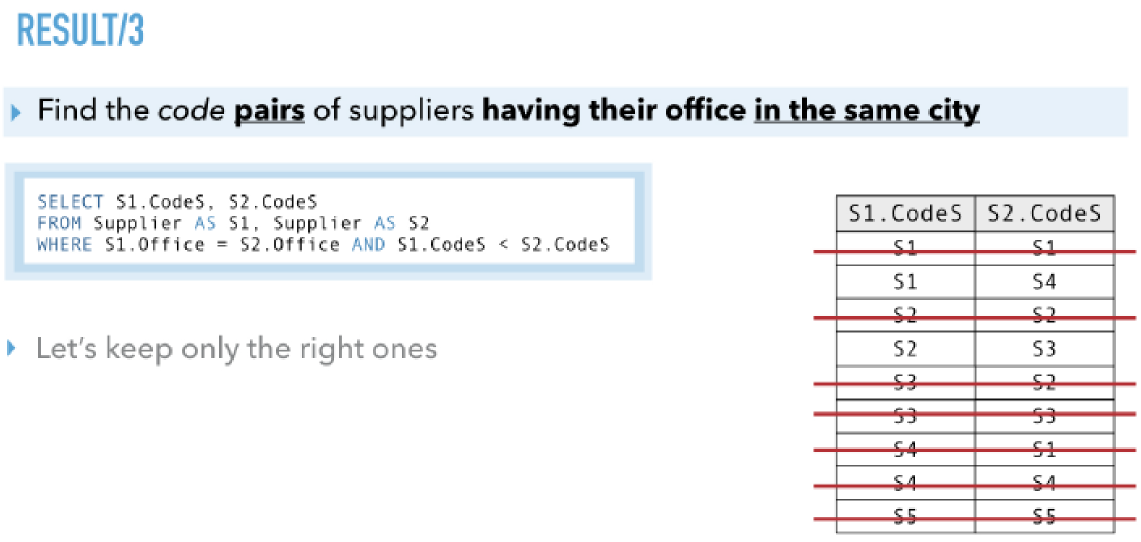
In the example we want to find *the name of suppliers of **at least** one red product*. Notice how we join the Products, Supply, and Supplier tables by joining on their CodeS and CodeP attributes.

Pay attention to the original query, and on its implementation. By asking, *"at least one product"*, we mean that we are satisfied with result sets containing suppliers of 2, 3, any red products. We are not able, however, to specify a condition where we want to retrieve suppliers that supply **EXACTLY** one product. We will see later how to specify such query.

8.7.1 Example Query in DB3

The query below retrieves the list of movies (title) having at least one female cast member.

```
SELECT title
FROM title_100k, cast_info_100k, person_100k
WHERE title_100k.id = cast_info_100k.movie_id
AND person_100k.id = cast_info_100k.person_id
AND gender = 'f'
```



RESULT/3

- Find the *code* **pairs** of suppliers **having their office in the same city**

```
SELECT S1.CodeS, S2.CodeS
FROM Supplier AS S1, Supplier AS S2
WHERE S1.Office = S2.Office AND S1.CodeS < S2.CodeS
```

- Let's keep only the right ones

S1.CodeS	S2.CodeS
~~S1~~	~~S1~~
S1	S4
~~S2~~	~~S2~~
S2	S3
~~S3~~	~~S2~~
~~S3~~	~~S3~~
~~S4~~	~~S1~~
~~S4~~	~~S4~~
~~S5~~	~~S5~~

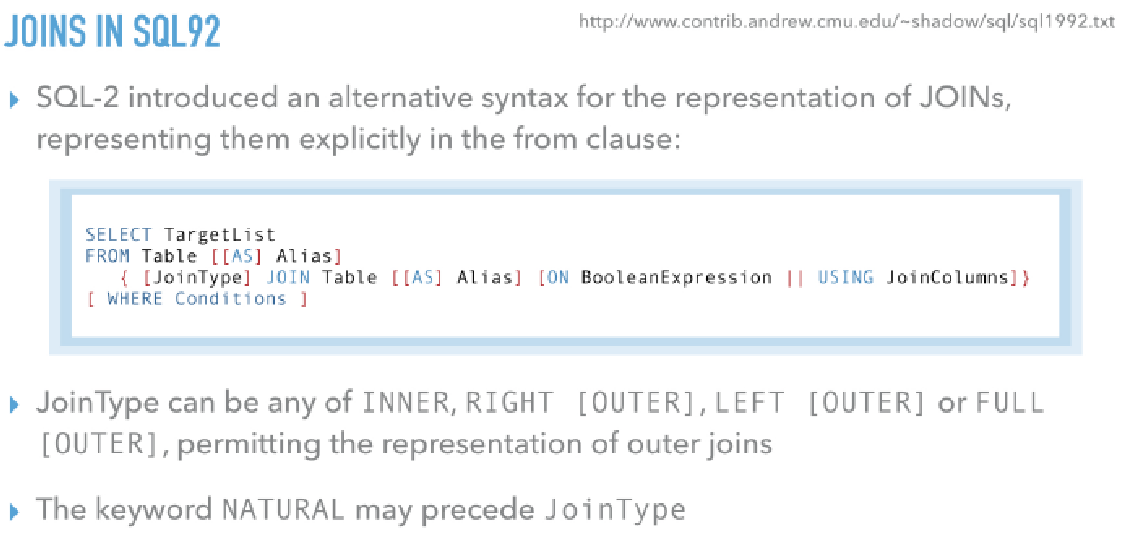
Figure 46: Slide 62 - Result

So, we can also define a condition that allows us to remove both issues. By imposing an order on the two codes (e.g. for one to be bigger than the other), we can immediately eliminate tuples with same values, and tuples in different orders.

8.11.1 Example Query in DB3

Now with `p1.id < p2.id` we obtained the desired results.

```
SELECT p1.name, p2.name, c1.movie_id
FROM person_100k AS p1, cast_info_100k AS c1, cast_info_100k AS c2,
   person_100k AS p2
WHERE p1.id = c1.person_id AND p2.id = c2.person_id AND c1.movie_id = c2.
   movie_id AND p1.id < p2.id
```



JOINS IN SQL92

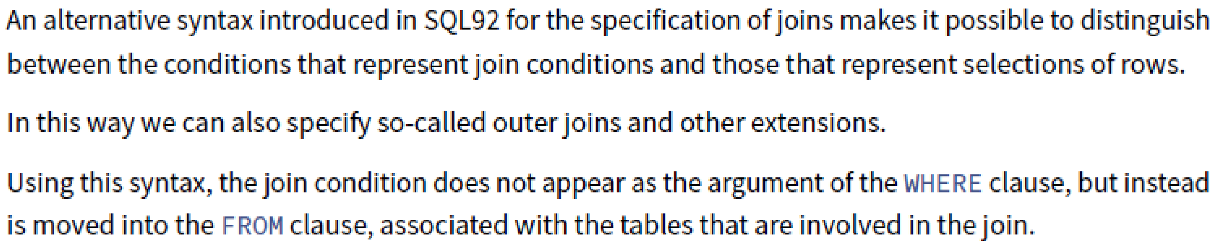
http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

- SQL-2 introduced an alternative syntax for the representation of JOINs, representing them explicitly in the from clause:

```
SELECT TargetList
FROM Table [[AS] Alias]
   { [JoinType] JOIN Table [[AS] Alias] [ON BooleanExpression || USING JoinColumns]}
[ WHERE Conditions ]
```

- JoinType can be any of INNER, RIGHT [OUTER], LEFT [OUTER] or FULL [OUTER], permitting the representation of outer joins
- The keyword NATURAL may precede JoinType

Figure 47: Slide 63 - JOINs in SQL92



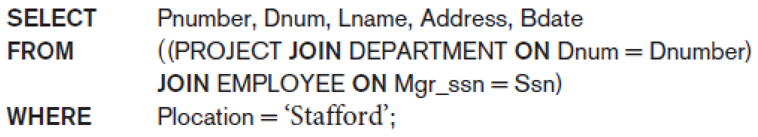
An alternative syntax introduced in SQL92 for the specification of joins makes it possible to distinguish between the conditions that represent join conditions and those that represent selections of rows.

In this way we can also specify so-called outer joins and other extensions.

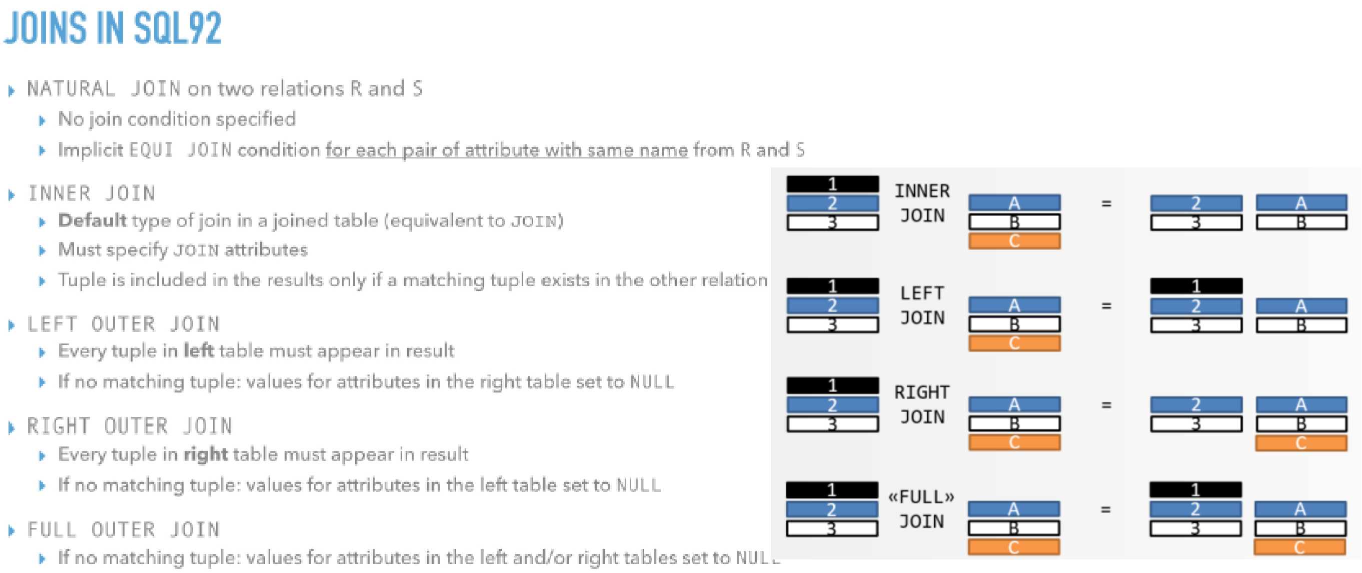
Using this syntax, the join condition does not appear as the argument of the WHERE clause, but instead is moved into the FROM clause, associated with the tables that are involved in the join.

The keyword CROSS JOIN is used to specify the CARTESIAN PRODUCT operation although this should be used only with the utmost care because it generates all possible tuple combinations, some of which may not exist in the real tables and thus wrong data!

It is also possible to nest join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a multiway join.



```
SELECT     Pnumber, Dnum, Lname, Address, Bdate
FROM       ((PROJECT JOIN DEPARTMENT ON Dnum = Dnumber)
           JOIN EMPLOYEE ON Mgr_ssn = Ssn)
WHERE      Plocation = 'Stafford';
```



JOINS IN SQL92

- NATURAL JOIN on two relations R and S
  - No join condition specified
  - Implicit EQUI JOIN condition for each pair of attribute with same name from R and S
- INNER JOIN
  - **Default** type of join in a joined table (equivalent to JOIN)
  - Must specify JOIN attributes
  - Tuple is included in the results only if a matching tuple exists in the other relation
- LEFT OUTER JOIN
  - Every tuple in **left** table must appear in result
  - If no matching tuple: values for attributes in the right table set to NULL
- RIGHT OUTER JOIN
  - Every tuple in **right** table must appear in result
  - If no matching tuple: values for attributes in the left table set to NULL
- FULL OUTER JOIN
  - If no matching tuple: values for attributes in the left and/or right tables set to NULL

Figure 48: Slide 64 - JOINs in SQL92

The parameter *JoinType* specifies which type of join to use, and for this we can substitute the terms `INNER`, `RIGHT OUTER`, `LEFT OUTER`, or `FULL OUTER` (the clause `OUTER` is optional).

The join condition is specified with the `ON` or `USING` clauses, or implicitly by the clause `NATURAL`. The join condition determines which rows from the two source tables are considered to "match".

The following explanations are partially taken from the the PostgreSQL documentation.

The `ON` clause is the most general kind of join condition: it takes a Boolean value expression of the same kind as is used in a `WHERE` clause. A pair of rows from a table `A` and `B` match if the `ON` expression evaluates to true.

The `USING` clause is a shorthand that allows you to take advantage of the specific situation where both sides of the join use the same name for the joining column(s). It takes a comma-separated list of the shared column names and forms a join condition that includes an equality comparison for each one. For example, joining `A` and `B` with `USING (a, b)` produces the join condition ON `A.a = B.a AND A.b = B.b`

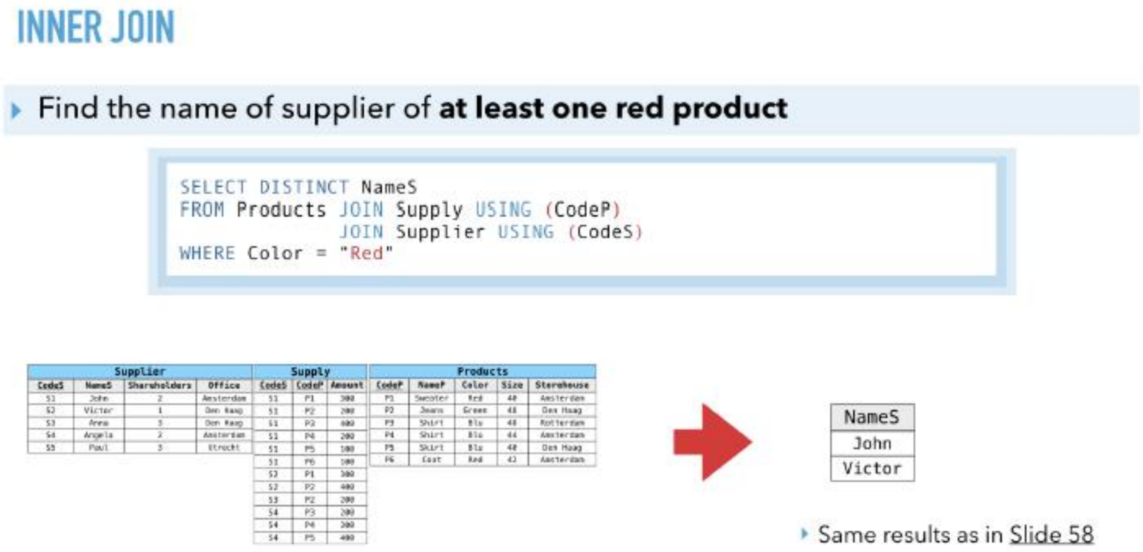
`NATURAL` is a shorthand form of `USING`: it forms a `USING` list consisting of all column names that appear in both input tables. As with `USING`, these columns appear only once in the output table. If there are no common column names, `NATURAL JOIN` behaves like `JOIN ... ON TRUE`, producing a cartesian product. Furthermore, the output of `JOIN ... USING` suppresses redundant columns: there is no need to print both of the matched columns, since they must have equal values. While `JOIN ... ON` produces all columns from `A` followed by all columns from ~`B`, `JOIN ... USING` produces one output column for each of the listed column pairs (in the listed order), followed by any remaining columns from `A`, followed by any remaining columns from `B`.

In spite of the advantage of an increased compactness, queries using natural joins can introduce risks to the applications, because its behaviour can change significantly as a result of small variations on the schema. Another reason is that the natural join makes it necessary to analyse completely the schema of the tables involved in order to understand the join condition. This is a disadvantage when writing and when reading the query, because in both situations it is necessary to do a careful comparison of the schemas of the joined tables in order to be sure of the behaviour of the query.

With the `INNER` join between the two tables, the rows involved in the join are generally a subset of the rows of each table. It can happen that some rows are not included because there exists no corresponding row in the other table for which the condition is satisfied. This property often conflicts with the demands of applications that might need to retain the rows that would be eliminated by the join. In writing the application, we might prefer to use null values to represent the absence of information in the other table. This is the role of `OUTER` joins.

There are three different types of `OUTER JOIN`: `LEFT`, `RIGHT`, and`FULL`.

The `LEFT` join gives the same result as the inner join, but includes the rows of the table that appears in the left of the join for which no corresponding rows exist in the right-hand table. The `RIGHT` join behaves symmetrically (keeps the rows of the right-hand table); finally, the `FULL` join gives the result of the inner join along with the rows excluded from both tables.



INNER JOIN

- Find the name of supplier of **at least one red product**

```
SELECT DISTINCT NameS
FROM Products JOIN Supply USING (CodeP)
              JOIN Supplier USING (CodeS)
WHERE Color = "Red"
```

Supplier			
CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supply		
CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

Products				
CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	48	Den Haag
P6	Coat	Red	42	Amsterdam

NameS
John
Victor

- Same results as in Slide 58

Figure 49: Slide 65 - Inner JOIN

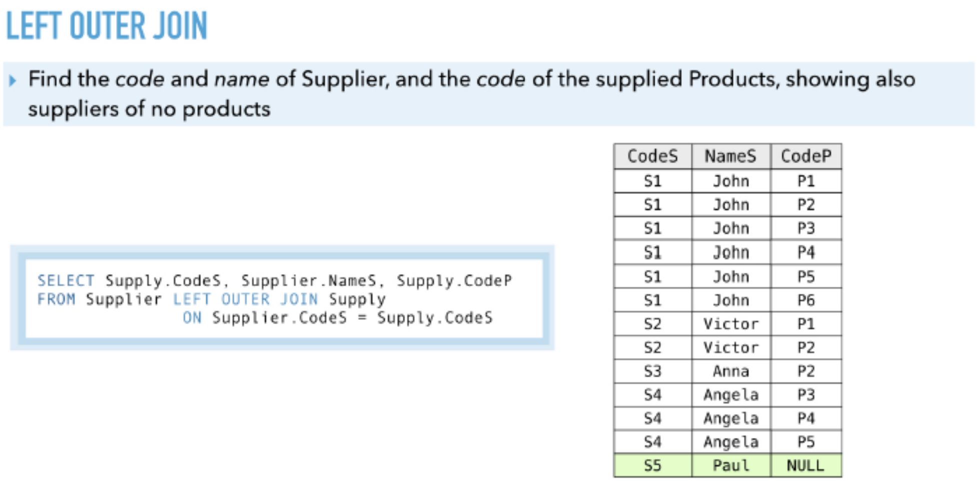
In figure, we show how one of the query previously written can be expressed using the `JOIN` syntax.

8.13.1 Example Query in DB3

Here are two different formulations of the same query (find the title of all the movies where `'Keanu Reeves'` had a role).

```
SELECT title
FROM title_100k, cast_info_100k, person_100k
WHERE title_100k.id = cast_info_100k.movie_id
    AND person_100k.id = cast_info_100k.person_id
    AND person_100k.name = 'Reeves, Keanu'
```

```
SELECT title
FROM title_100k JOIN cast_info_100k ON title_100k.id = cast_info_100k.
   movie_id
    JOIN person_100k ON person_100k.id = cast_info_100k.person_id
WHERE person_100k.name = 'Reeves, Keanu'
```



LEFT OUTER JOIN

- Find the *code* and *name* of Supplier, and the *code* of the supplied Products, showing also suppliers of no products

```
SELECT Supply.CodeS, Supplier.NameS, Supply.CodeP
FROM Supplier LEFT OUTER JOIN Supply
              ON Supplier.CodeS = Supply.CodeS
```

CodeS	NameS	CodeP
S1	John	P1
S1	John	P2
S1	John	P3
S1	John	P4
S1	John	P5
S1	John	P6
S2	Victor	P1
S2	Victor	P2
S3	Anna	P2
S4	Angela	P3
S4	Angela	P4
S4	Angela	P5
S5	Paul	NULL

Figure 50: Slide 66 - Left Outer JOIN

Here instead, we exemplify the use of a `LEFT OUTER JOIN`, on the tables `Supply` and `Supplier`. You can notice how the tuple `(S5, Paul,NULL)` is returned even if such supply has no supplier.

Alter Tables

Add column

ALTER TABLE products ADD COLUMN description text;

Drop column

ALTER TABLE products DROP COLUMN description CASCADE;

Add constraint

ALTER TABLE products ADD CHECK (name <> '');

ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);

ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;

ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;

Drop constraint

ALTER TABLE products DROP CONSTRAINT some_name;

Change default, data type

ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;

ALTER TABLE products ALTER COLUMN price DROP DEFAULT;

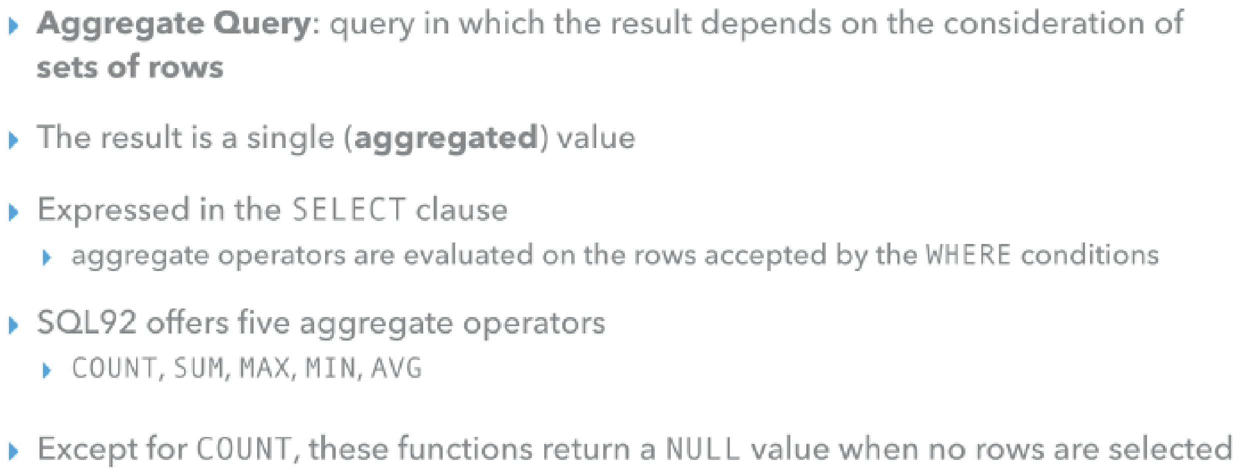
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);

Renaming column, table

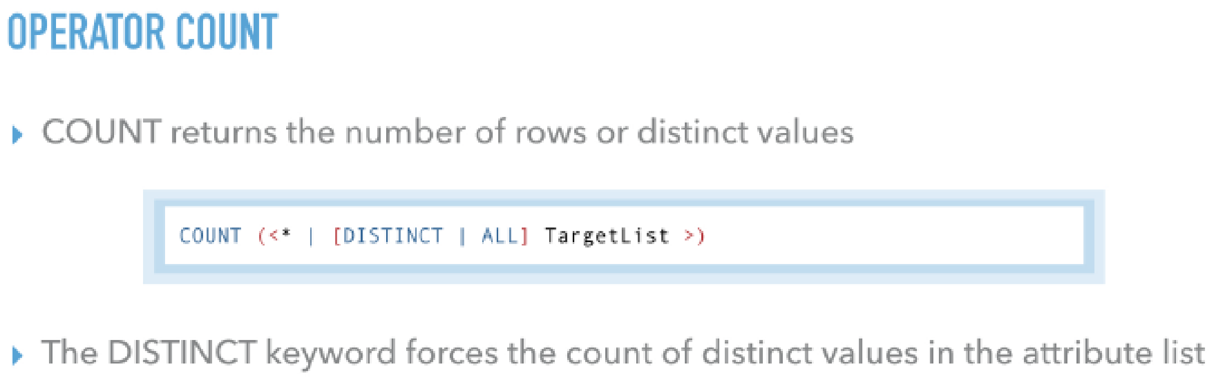
ALTER TABLE products RENAME COLUMN product_no TO product_number;

ALTER TABLE products RENAME TO items;

Lecture 8



- **Aggregate Query**: query in which the result depends on the consideration of **sets of rows**
- The result is a single (**aggregated**) value
- Expressed in the SELECT clause
  - aggregate operators are evaluated on the rows accepted by the WHERE conditions
- SQL92 offers five aggregate operators
  - COUNT, SUM, MAX, MIN, AVG
- Except for COUNT, these functions return a NULL value when no rows are selected

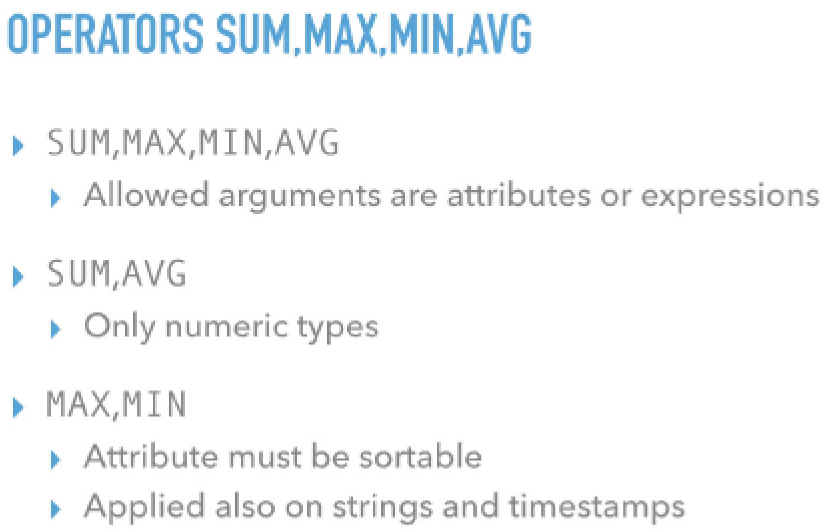


OPERATOR COUNT

- COUNT returns the number of rows or distinct values

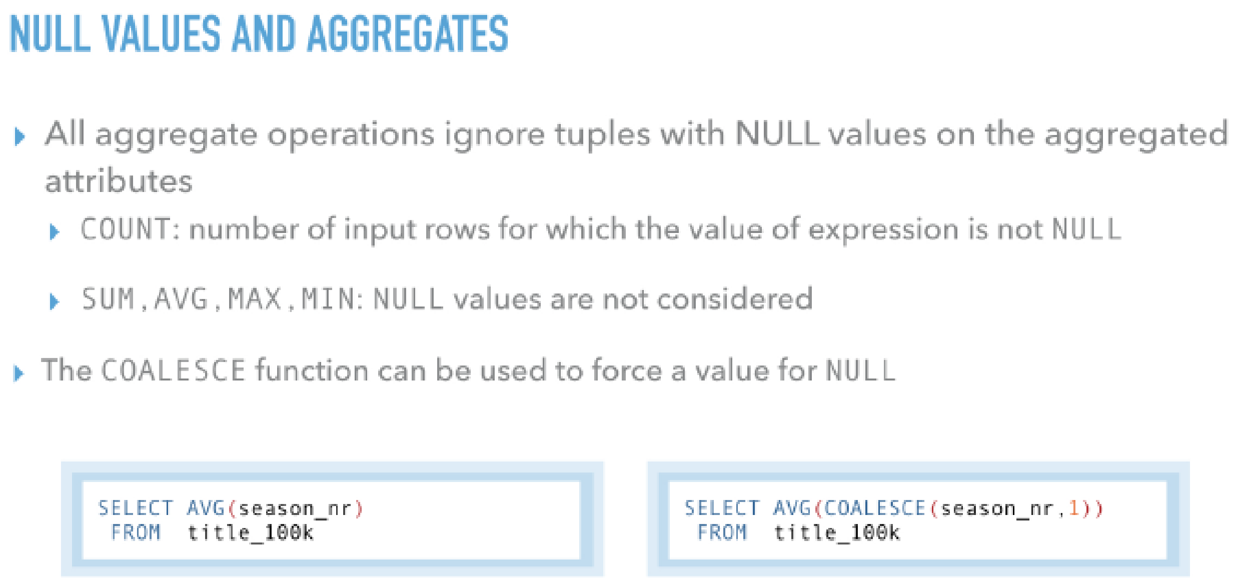
```
COUNT (<* | [DISTINCT | ALL] TargetList >)
```

- The DISTINCT keyword forces the count of distinct values in the attribute list



OPERATORS SUM,MAX,MIN,AVG

- SUM,MAX,MIN,AVG
  - Allowed arguments are attributes or expressions
- SUM,AVG
  - Only numeric types
- MAX,MIN
  - Attribute must be sortable
  - Applied also on strings and timestamps



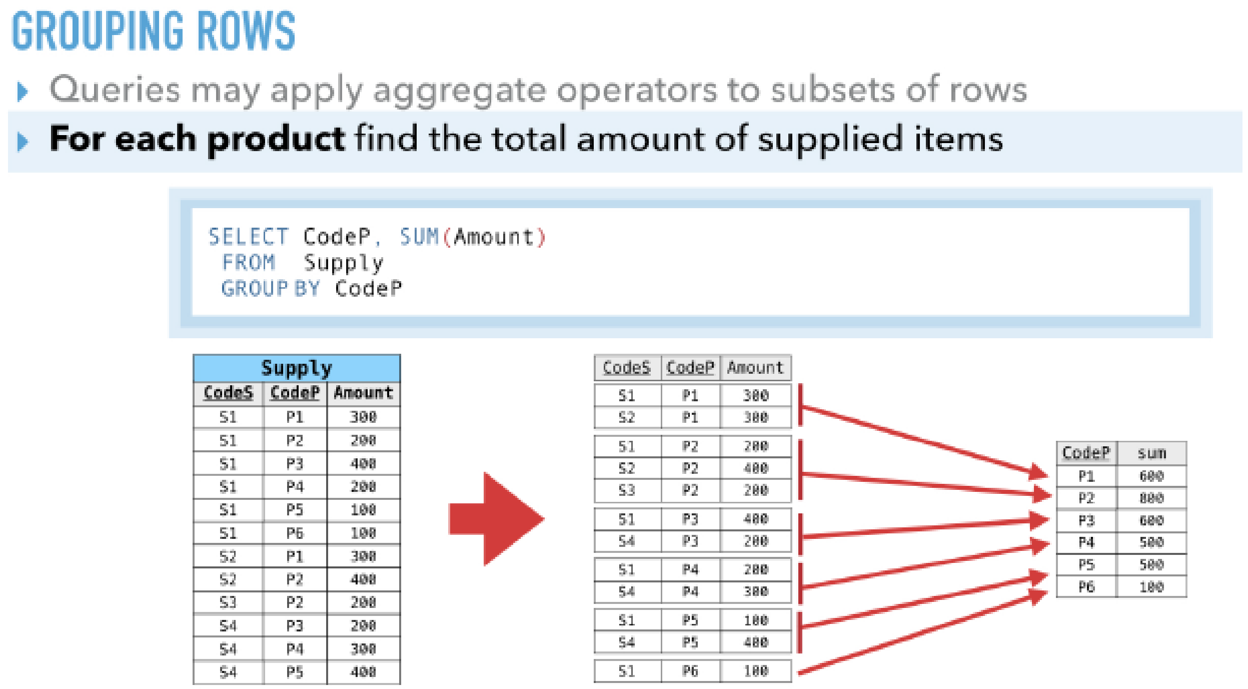
NULL VALUES AND AGGREGATES

- All aggregate operations ignore tuples with NULL values on the aggregated attributes
  - COUNT: number of input rows for which the value of expression is not NULL
  - SUM, AVG, MAX, MIN: NULL values are not considered
- The COALESCE function can be used to force a value for NULL

```
SELECT AVG(season_nr)
 FROM  title_100k
```

```
SELECT AVG(COALESCE(season_nr,1))
 FROM  title_100k
```

To use an aggregate function it must be explicitly stated to which subset of rows it must adhere to. That is achieved with the **GROUP BY clause**. The clause accepts as argument a set of attributes, **the query will operate separately on each set of rows that possess the same values for these set of attributes**.



GROUPING ROWS

- Queries may apply aggregate operators to subsets of rows
- **For each product** find the total amount of supplied items

```
SELECT CodeP, SUM(Amount)
 FROM  Supply
 GROUP BY CodeP
```

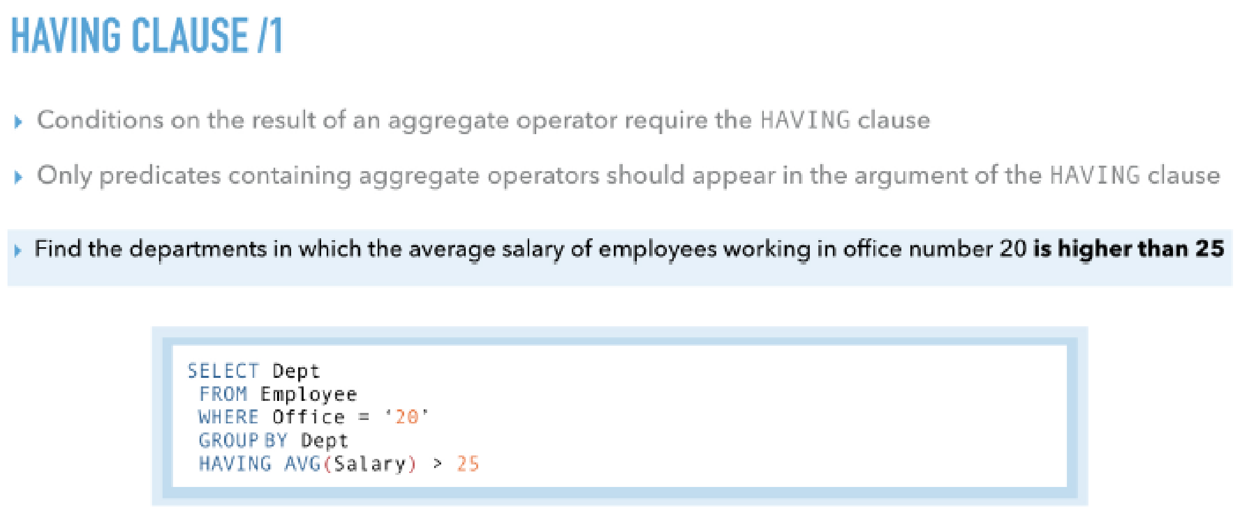
Supply		
CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

CodeS	CodeP	Amount
S1	P1	300
S2	P1	300
S1	P2	200
S2	P2	400
S3	P2	200
S1	P3	400
S4	P3	200
S1	P4	200
S4	P4	300
S1	P5	100
S4	P5	400
S1	P6	100

CodeP	sum
P1	600
P2	800
P3	600
P4	500
P5	500
P6	100

Once the rows are partitioned into subsets, **the aggregate operator is applied separately to each subset**. **Each separate result is reported as a corresponging row in the result query**.

The attributes that can appear in the select clause must be a subset of the attributes used in the GROUP BY clause. Usually it is the list of all selected attributes (except the aggregate one).



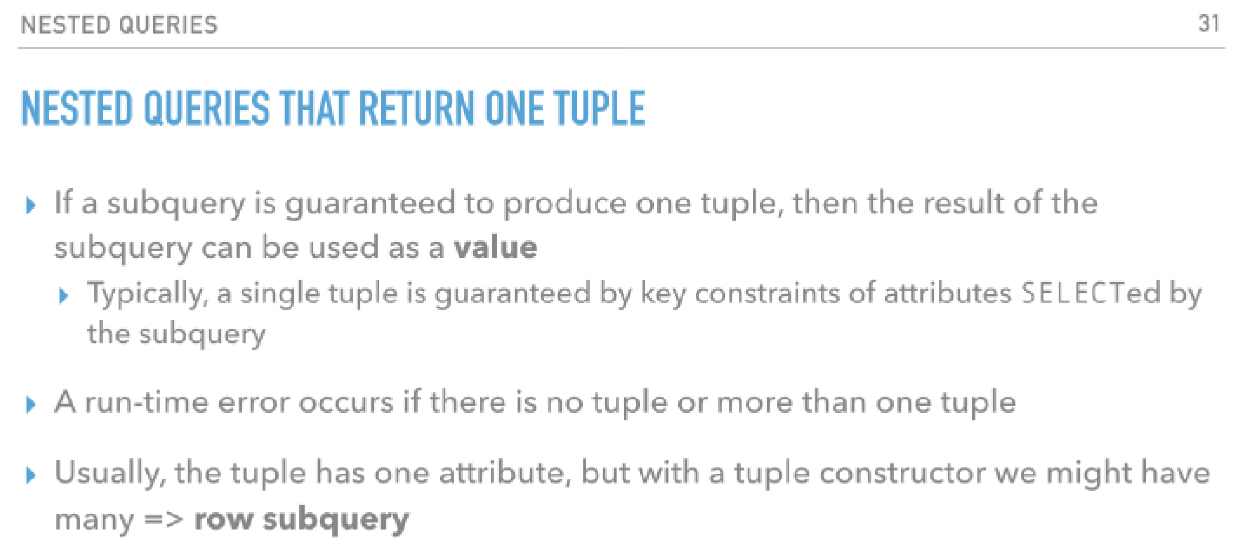
HAVING CLAUSE /1

- Conditions on the result of an aggregate operator require the HAVING clause
- Only predicates containing aggregate operators should appear in the argument of the HAVING clause

- Find the departments in which the average salary of employees working in office number 20 **is higher than 25**

```
SELECT Dept
 FROM Employee
 WHERE Office = ‘20’
 GROUP BY Dept
 HAVING AVG(Salary) > 25
```

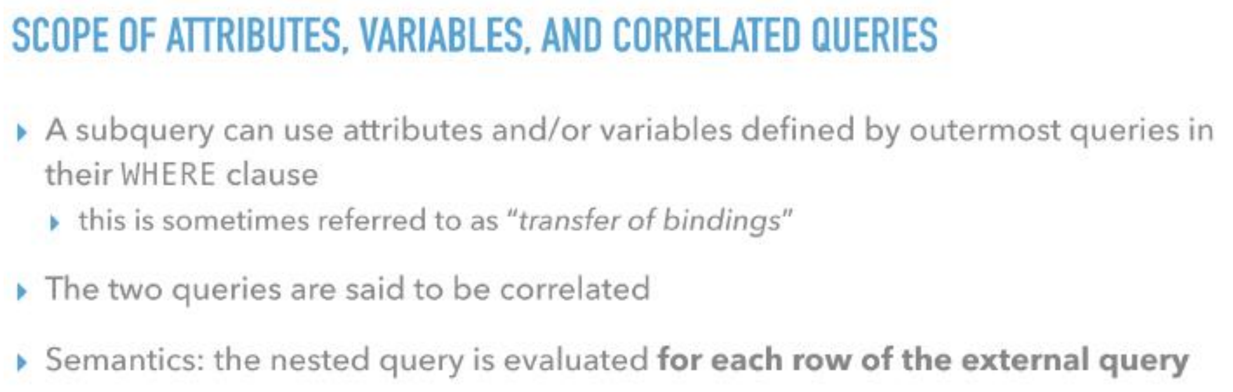
The syntax also allows for the definition of queries using the HAVING clause, without a corresponding GROUP BY clause. The having clause will also accept as argument a boolean expression of simple predicates. The simple predicates are generally comparisons between the result of the evaluation of an aggregate operator and a generic expression.



NESTED QUERIES 31

NESTED QUERIES THAT RETURN ONE TUPLE

- If a subquery is guaranteed to produce one tuple, then the result of the subquery can be used as a **value**
  - Typically, a single tuple is guaranteed by key constraints of attributes SELECTed by the subquery
- A run-time error occurs if there is no tuple or more than one tuple
- Usually, the tuple has one attribute, but with a tuple constructor we might have many => **row subquery**

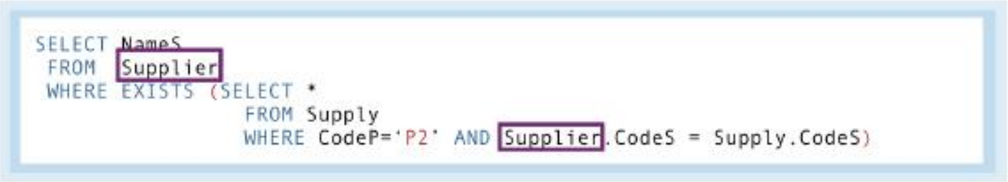


SCOPE OF ATTRIBUTES, VARIABLES, AND CORRELATED QUERIES

- A subquery can use attributes and/or variables defined by outermost queries in their WHERE clause
  - this is sometimes referred to as *“transfer of bindings”*
- The two queries are said to be correlated
- Semantics: the nested query is evaluated **for each row of the external query**

THE EXISTS OPERATOR/1

- Find the name of the suppliers that supplied P2 at least once



```
SELECT NameS
 FROM  Supplier
 WHERE EXISTS (SELECT *
                FROM Supply
                WHERE CodeP='P2' AND Supplier.CodeS = Supply.CodeS)
```

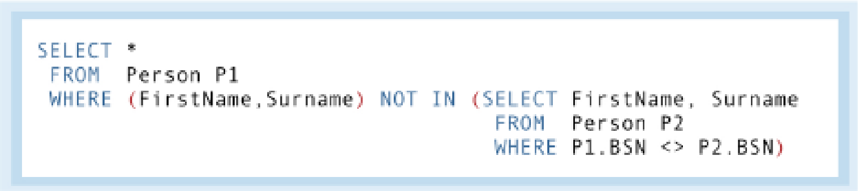
- !!! We need to test the existence of a supply for the evaluation of suppliers
- `Suppliers.CodeS = Supply.CodeS` imposes a correlation between external and internal query

LIMITATIONS

- A query cannot refer to attributes in a subquery, or in a query at the same level of nesting

TUPLE CONSTRUCTOR

- The comparison with the nested query may involve more than one attributes
- The attributes must be enclosed within a pair of curved brackets (tuple constructor)
- The query in the previous slide can be expressed as:



```
SELECT *
 FROM  Person P1
 WHERE (FirstName,Surname) NOT IN (SELECT FirstName, Surname
                                    FROM  Person P2
                                    WHERE P1.BSN <> P2.BSN)
```

SET QUERIES

- Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:
  - `(subquery)INTERSECT[ALL](subquery)`
  - `(subquery)EXCEPT[ALL](subquery)`

CAN BE EXPRESSED WITH OTHER OPERATORS (TYPICALLY SUB-QUERIES)

  - `(subquery)UNION[ALL](subquery)`

ENHANCEMENT OF THE EXPRESSIVE POWER

SET SEMANTIC OF SET QUERIES

- Although the `SELECT-FROM-WHERE` statement uses **bag semantics**, the *default* for union, intersection, and difference is **set semantics**.
  - That is, duplicates are eliminated as the operation is applied

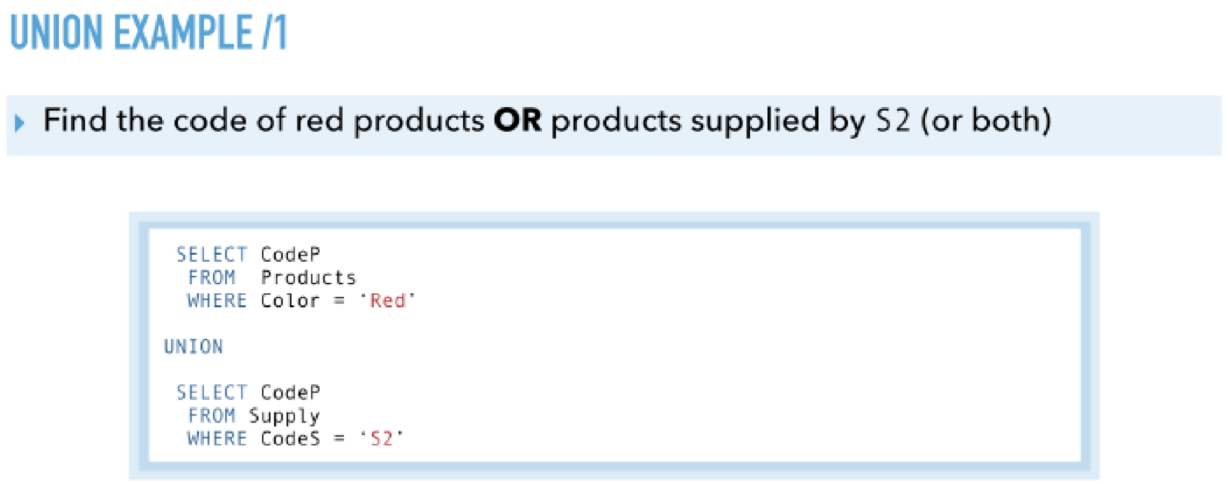
- Motivation: Efficiency
  - When projecting attributes, it is easier to avoid **eliminating duplicates**. Just work tuple-at-a-time.
  - When doing intersection or difference, it is most efficient to **sort the relations first** At that point you may as well eliminate the duplicates anyway

UNION

- A single `SELECT` cannot represent unions of values from two or more tables

```
A UNION [ALL] B
```

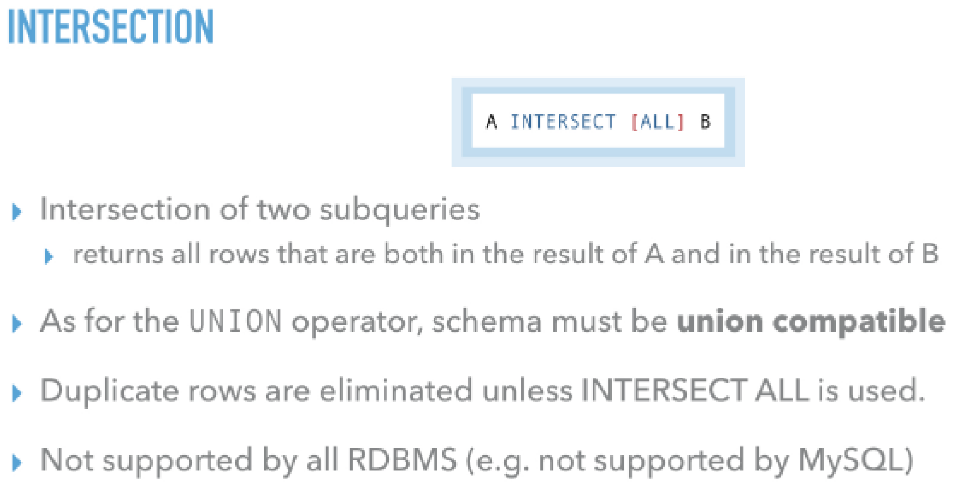
- It executes the union of two relational expressions
  - Expressions generated by `SELECT` clauses
  - Table `A` and Table `B` must be **union compatible**
    - i.e. have compatible schema
    - same number of output fields, in the same order, and with the same or compatible data types
- Duplicate removal
  - `UNION` removes duplicates
  - `UNION ALL` does not remove duplicates



UNION EXAMPLE /1

- Find the code of red products **OR** products supplied by S2 (or both)

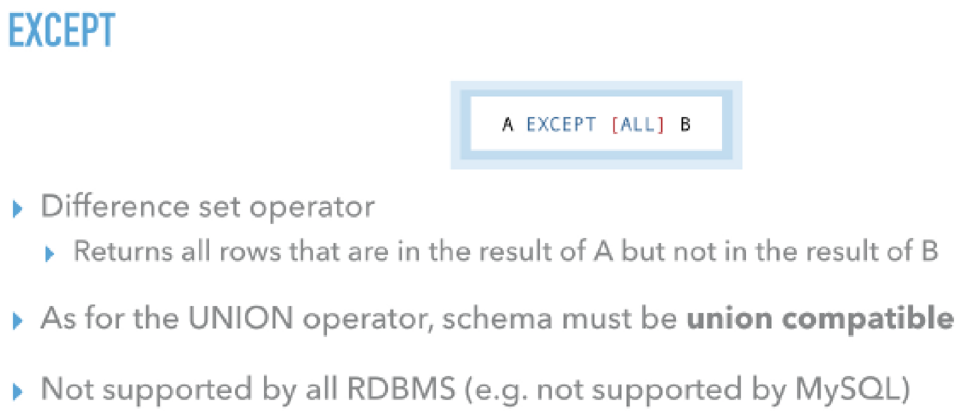
```
 SELECT CodeP
  FROM  Products
  WHERE Color = 'Red'

UNION

 SELECT CodeP
  FROM Supply
  WHERE CodeS = 'S2'
```



INTERSECTION

```
A INTERSECT [ALL] B
```

- Intersection of two subqueries
  - returns all rows that are both in the result of A and in the result of B
- As for the UNION operator, schema must be **union compatible**
- Duplicate rows are eliminated unless INTERSECT ALL is used.
- Not supported by all RDBMS (e.g. not supported by MySQL)

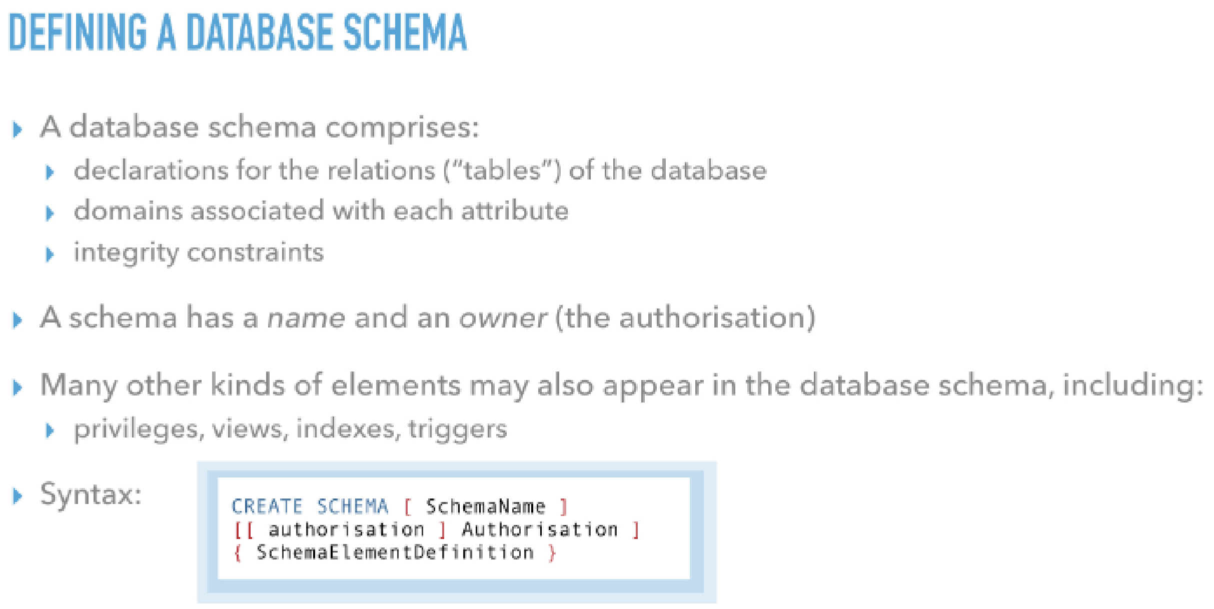


EXCEPT

```
A EXCEPT [ALL] B
```

- Difference set operator
  - Returns all rows that are in the result of A but not in the result of B
- As for the UNION operator, schema must be **union compatible**
- Not supported by all RDBMS (e.g. not supported by MySQL)

Use of SQL for the definition of database schemas (DDL)



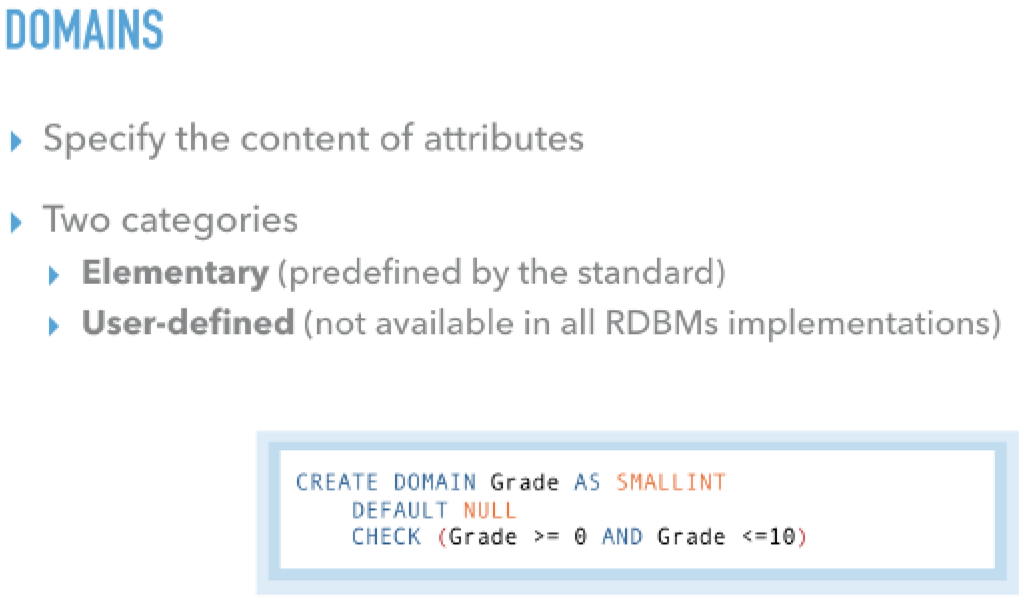
DEFINING A DATABASE SCHEMA

- A database schema comprises:
  - declarations for the relations ("tables") of the database
  - domains associated with each attribute
  - integrity constraints
- A schema has a *name* and an *owner* (the authorisation)
- Many other kinds of elements may also appear in the database schema, including:
  - privileges, views, indexes, triggers
- Syntax:

```
CREATE SCHEMA [ SchemaName ]
[[ authorisation ] Authorisation ]
{ SchemaElementDefinition }
```

SQL makes it possible to define a database schema as a collection of objects; each schema consists of a set of domains and tables, defined by the syntax in the slide. A schema also includes indices, assertions, views and privileges. Authorization represents the name of the user who owns the schema. If the term is omitted, it is assumed that the user who issued the command is the owner. The name of the schema can be omitted and, in this case, the name of the owner is adopted as the name of the schema. After the create schema command, the user can define the schema components.

It is not necessary for all the components to be defined at the same time as the schema is created.

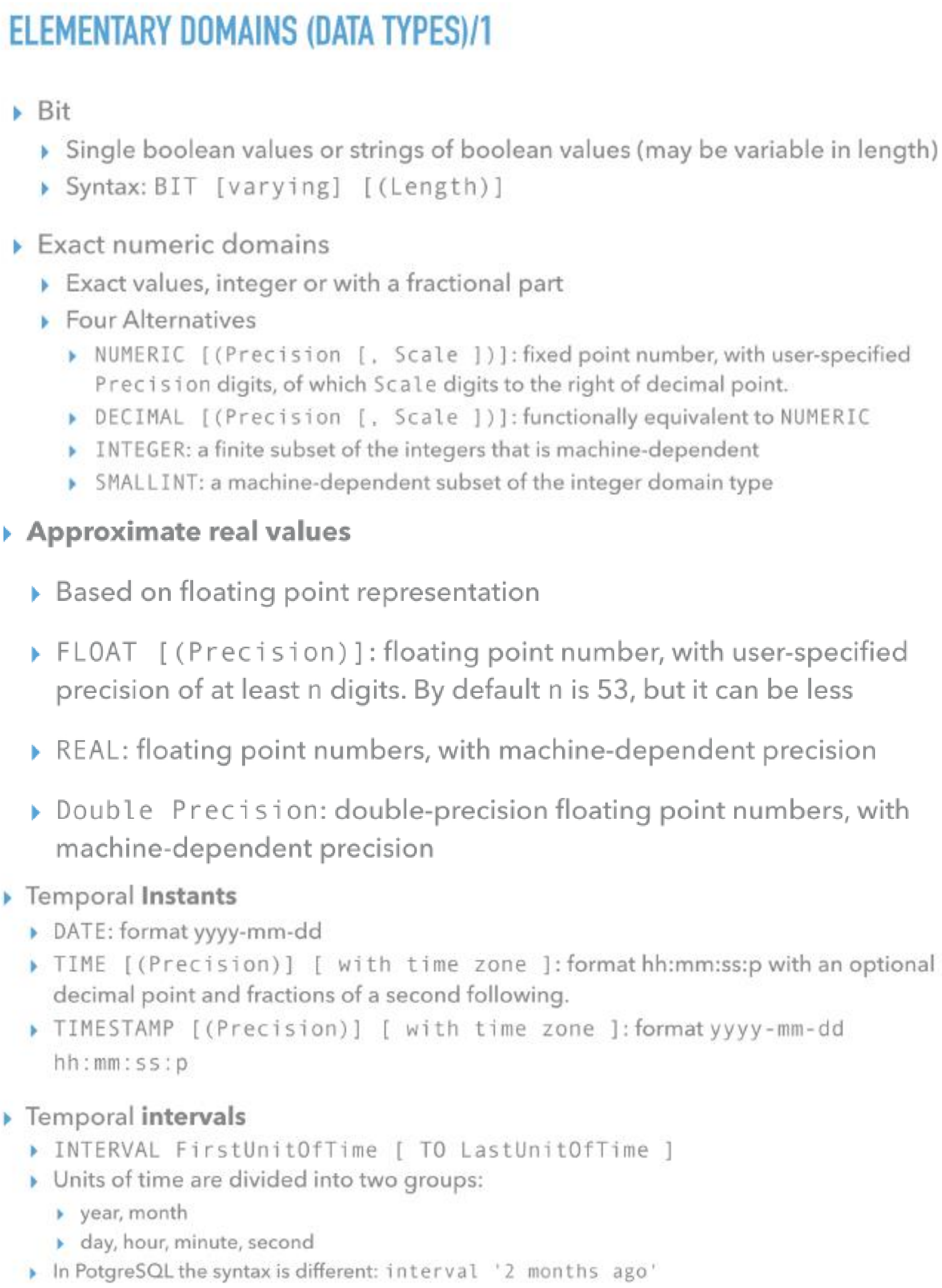


DOMAINS

- Specify the content of attributes
- Two categories
  - **Elementary** (predefined by the standard)
  - **User-defined** (not available in all RDBMs implementations)

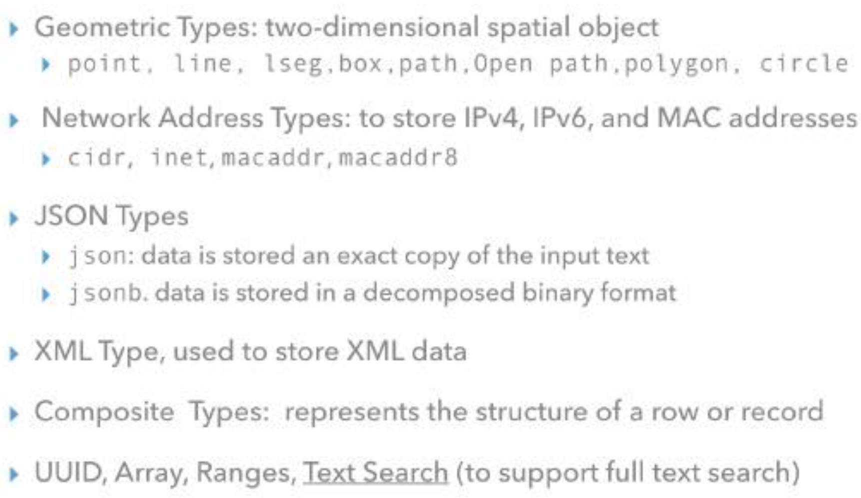
```
CREATE DOMAIN Grade AS SMALLINT
    DEFAULT NULL
    CHECK (Grade >= 0 AND Grade <=10)
```

A domain is essentially a data type with optional constraints (restrictions on the allowed set of values). The user who defines a domain becomes its owner.



ELEMENTARY DOMAINS (DATA TYPES)/1

- Bit
  - Single boolean values or strings of boolean values (may be variable in length)
  - Syntax: `BIT [varying] [(Length)]`
- Exact numeric domains
  - Exact values, integer or with a fractional part
  - Four Alternatives
    - `NUMERIC [(Precision [, Scale ])]`: fixed point number, with user-specified `Precision` digits, of which `Scale` digits to the right of decimal point.
    - `DECIMAL [(Precision [, Scale ])]`: functionally equivalent to `NUMERIC`
    - `INTEGER`: a finite subset of the integers that is machine-dependent
    - `SMALLINT`: a machine-dependent subset of the integer domain type
- **Approximate real values**
  - Based on floating point representation
  - `FLOAT [(Precision)]`: floating point number, with user-specified precision of at least n digits. By default n is 53, but it can be less
  - `REAL`: floating point numbers, with machine-dependent precision
  - `Double Precision`: double-precision floating point numbers, with machine-dependent precision
- Temporal **Instants**
  - `DATE`: format yyyy-mm-dd
  - `TIME [(Precision)] [ with time zone ]`: format hh:mm:ss:p with an optional decimal point and fractions of a second following.
  - `TIMESTAMP [(Precision)] [ with time zone ]`: format `yyyy-mm-dd hh:mm:ss:p`
- Temporal **intervals**
  - `INTERVAL FirstUnitOfTime [ TO LastUnitOfTime ]`
  - Units of time are divided into two groups:
    - year, month
    - day, hour, minute, second
  - In PotgreSQL the syntax is different: `interval '2 months ago'`



- Geometric Types: two-dimensional spatial object
  - `point, line, lseg,box,path,Open path,polygon, circle`
- Network Address Types: to store IPv4, IPv6, and MAC addresses
  - `cidr, inet,macaddr,macaddr8`
- JSON Types
  - `json`: data is stored an exact copy of the input text
  - `jsonb`. data is stored in a decomposed binary format
- XML Type, used to store XML data
- Composite Types: represents the structure of a row or record
- UUID, Array, Ranges, Text Search (to support full text search)

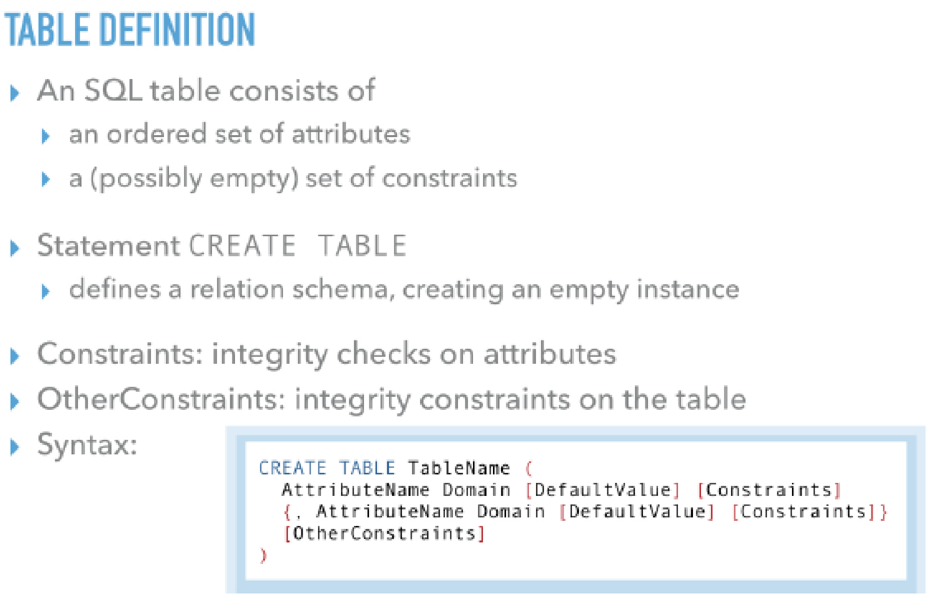
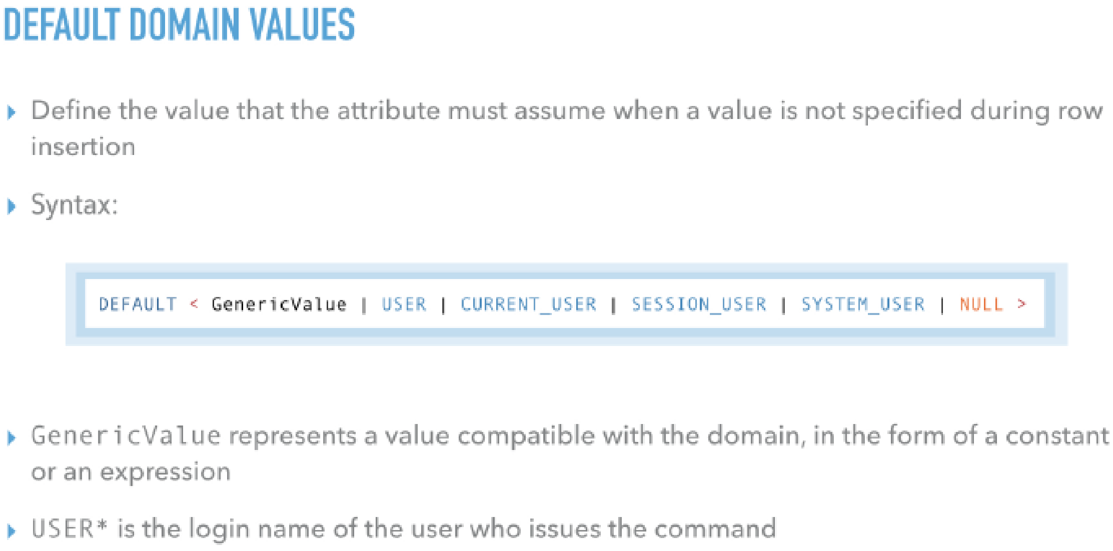


TABLE DEFINITION

- An SQL table consists of
  - an ordered set of attributes
  - a (possibly empty) set of constraints
- Statement `CREATE TABLE`
  - defines a relation schema, creating an empty instance
- Constraints: integrity checks on attributes
- OtherConstraints: integrity constraints on the table
- Syntax:

```
CREATE TABLE TableName (
  AttributeName Domain [DefaultValue] [Constraints]
  {, AttributeName Domain [DefaultValue] [Constraints]}
  [OtherConstraints]
)
```



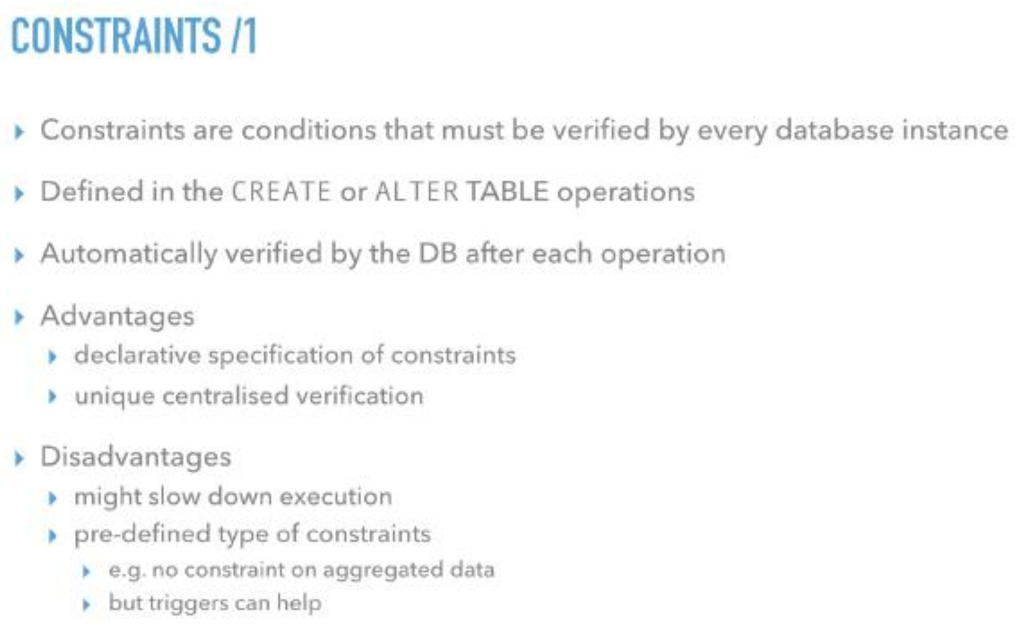
DEFAULT DOMAIN VALUES

- Define the value that the attribute must assume when a value is not specified during row insertion
- Syntax:

```
DEFAULT < GenericValue | USER | CURRENT_USER | SESSION_USER | SYSTEM_USER | NULL >
```

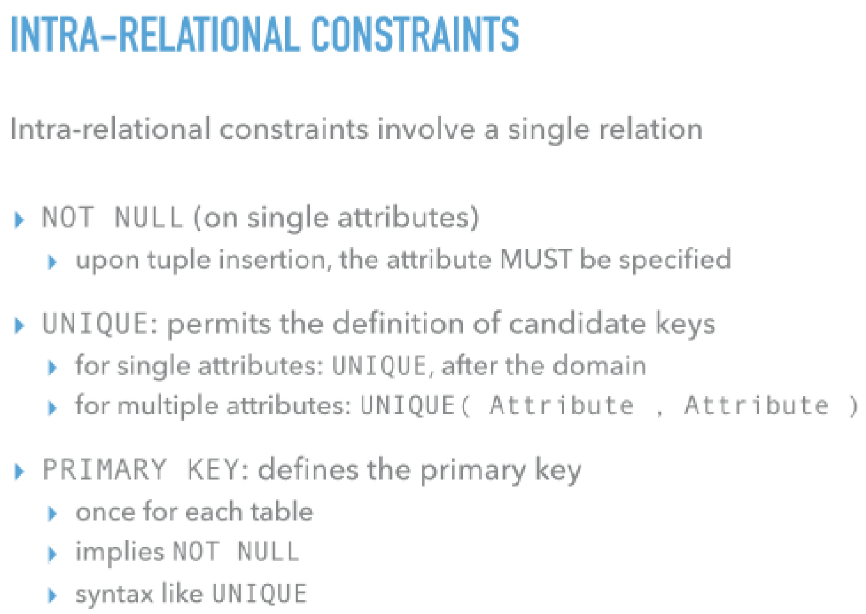
- `GenericValue` represents a value compatible with the domain, in the form of a constant or an expression
- `USER*` is the login name of the user who issues the command

A common example is for a timestamp column to have a default of CURRENT_TIMESTAMP, so that it gets set to the time of row insertion. Another common example is generating a “serial number” for each row. In PostgreSQL this is typically done by using the SERIAL data type.



CONSTRAINTS /1

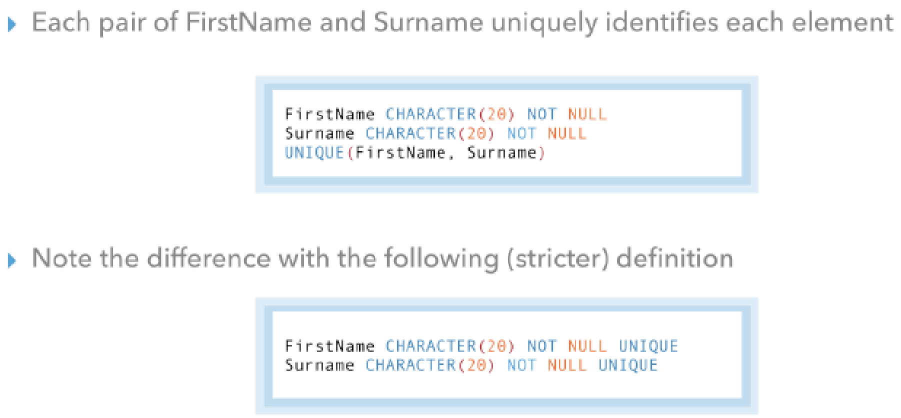
- Constraints are conditions that must be verified by every database instance
- Defined in the `CREATE` or `ALTER` TABLE operations
- Automatically verified by the DB after each operation
- Advantages
  - declarative specification of constraints
  - unique centralised verification
- Disadvantages
  - might slow down execution
  - pre-defined type of constraints
    - e.g. no constraint on aggregated data
    - but triggers can help



INTRA-RELATIONAL CONSTRAINTS

Intra-relational constraints involve a single relation

- `NOT NULL` (on single attributes)
  - upon tuple insertion, the attribute MUST be specified
- `UNIQUE`: permits the definition of candidate keys
  - for single attributes: `UNIQUE`, after the domain
  - for multiple attributes: `UNIQUE( Attribute , Attribute )`
- `PRIMARY KEY`: defines the primary key
  - once for each table
  - implies `NOT NULL`
  - syntax like `UNIQUE`



- Each pair of FirstName and Surname uniquely identifies each element

```
FirstName CHARACTER(20) NOT NULL
Surname CHARACTER(20) NOT NULL
UNIQUE(FirstName, Surname)
```

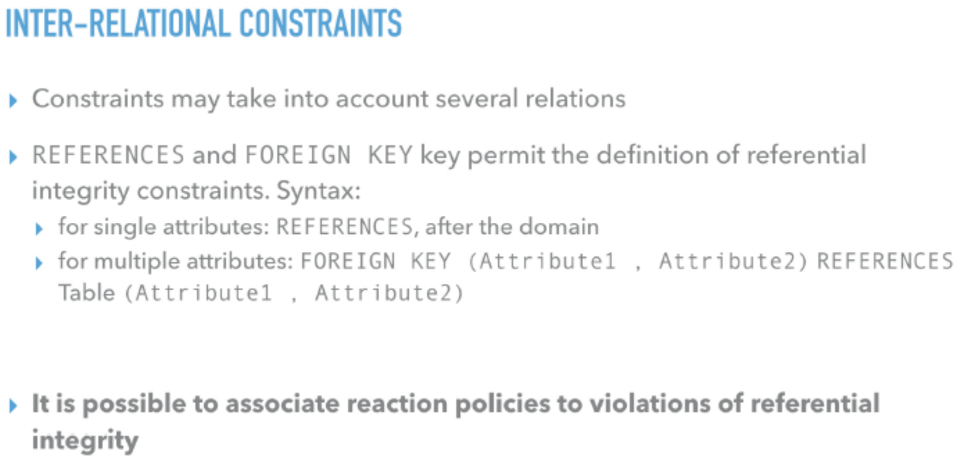
- Note the difference with the following (stricter) definition

```
FirstName CHARACTER(20) NOT NULL UNIQUE
Surname CHARACTER(20) NOT NULL UNIQUE
```

Figure 78: Slide 82 - Example of Intra-Relational Constraints

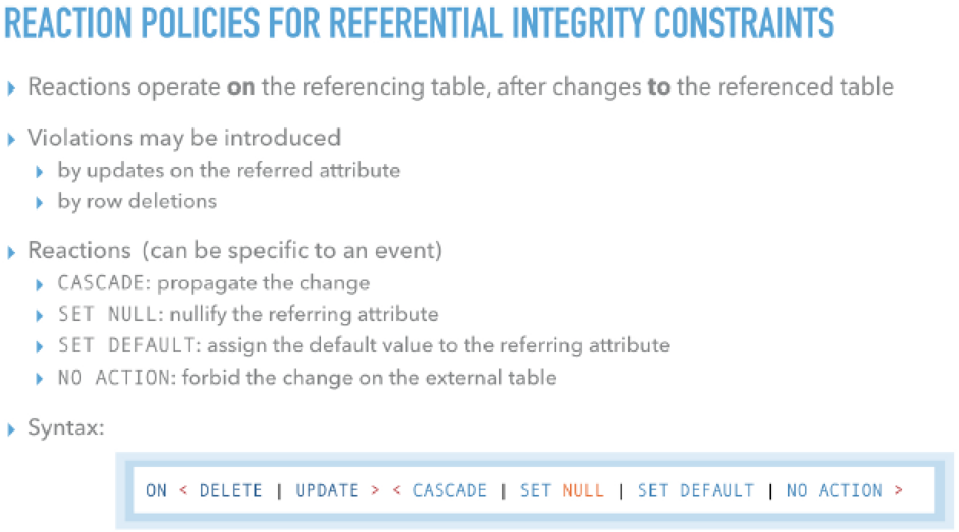
The slide contains show examples of usage for the `UNIQUE` constraint.

In the first case, the constraint imposes the condition that there can be no two rows that have both the same first name and the same surname. In the second (stricter) case, the constraint is violated if either the same first name or the same surname appears more that once.



INTER-RELATIONAL CONSTRAINTS

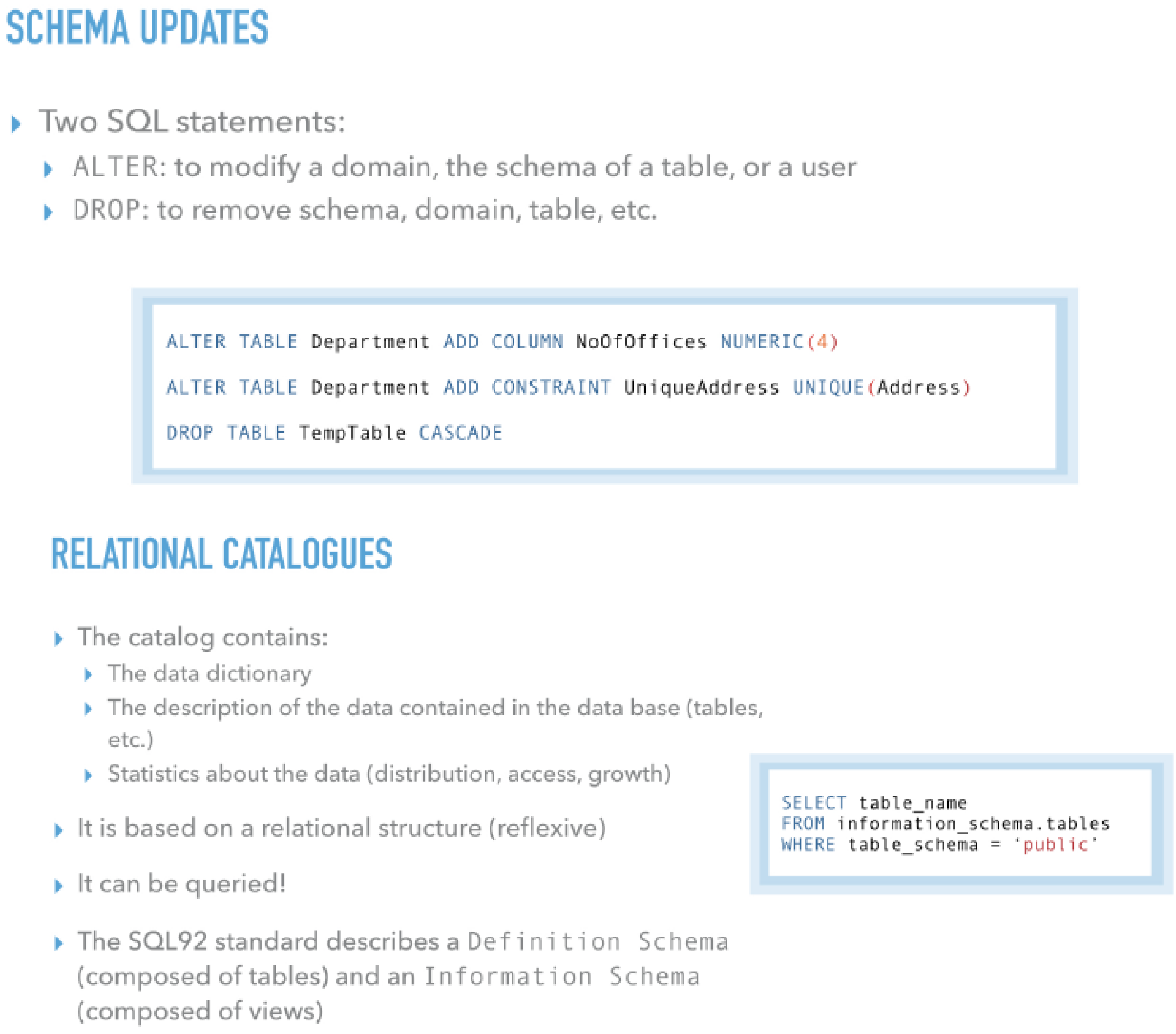
- Constraints may take into account several relations
- `REFERENCES` and `FOREIGN KEY` key permit the definition of referential integrity constraints. Syntax:
  - for single attributes: `REFERENCES`, after the domain
  - for multiple attributes: `FOREIGN KEY (Attribute1 , Attribute2) REFERENCES` Table `(Attribute1 , Attribute2)`
- **It is possible to associate reaction policies to violations of referential integrity**



REACTION POLICIES FOR REFERENTIAL INTEGRITY CONSTRAINTS

- Reactions operate **on** the referencing table, after changes **to** the referenced table
- Violations may be introduced
  - by updates on the referred attribute
  - by row deletions
- Reactions (can be specific to an event)
  - `CASCADE`: propagate the change
  - `SET NULL`: nullify the referring attribute
  - `SET DEFAULT`: assign the default value to the referring attribute
  - `NO ACTION`: forbid the change on the external table
- Syntax:

```
ON < DELETE | UPDATE > < CASCADE | SET NULL | SET DEFAULT | NO ACTION >
```



SCHEMA UPDATES

- Two SQL statements:
  - ALTER: to modify a domain, the schema of a table, or a user
  - DROP: to remove schema, domain, table, etc.

```
ALTER TABLE Department ADD COLUMN NoOfOffices NUMERIC(4)

ALTER TABLE Department ADD CONSTRAINT UniqueAddress UNIQUE(Address)

DROP TABLE TempTable CASCADE
```

RELATIONAL CATALOGUES

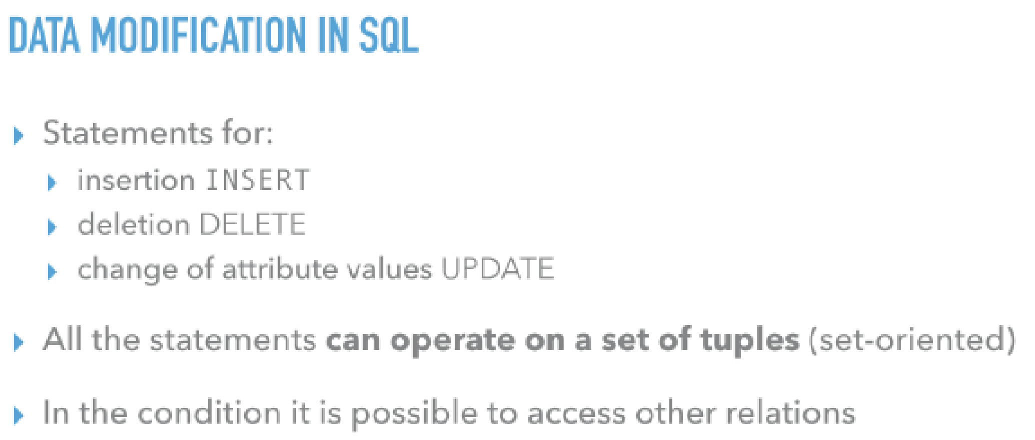
- The catalog contains:
  - The data dictionary
  - The description of the data contained in the data base (tables, etc.)
  - Statistics about the data (distribution, access, growth)
- It is based on a relational structure (reflexive)
- It can be queried!
- The SQL92 standard describes a `Definition Schema` (composed of tables) and an `Information Schema` (composed of views)

```
SELECT table_name
FROM information_schema.tables
WHERE table_schema = 'public'
```

Figure 82: Slide 87 - Relational Catalogues

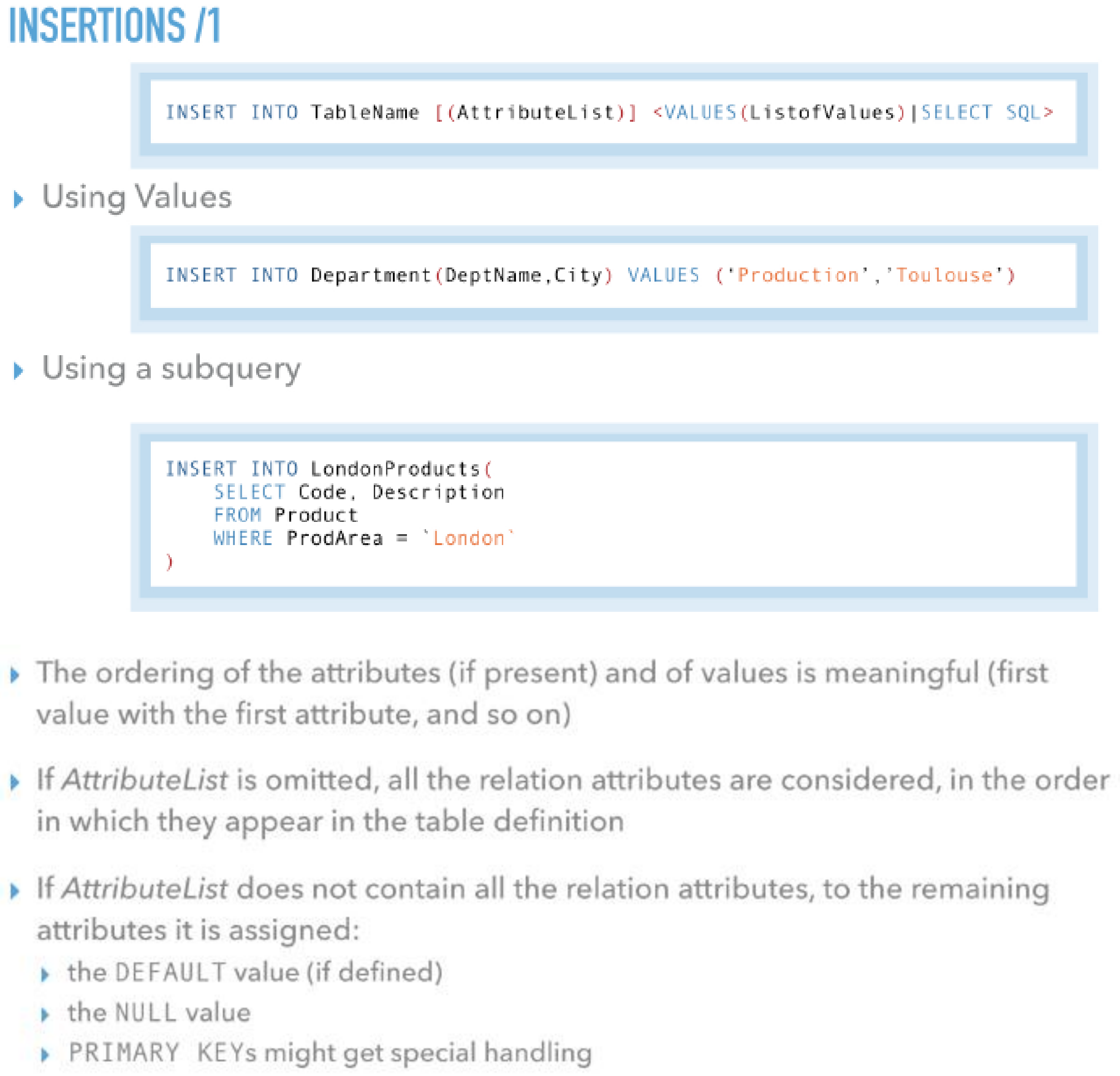
Although only partly specified by the standard, each relational DBMS manages its own **data dictionary** (or rather the description of the tables present in the database) using a relational schema. The database therefore contains two types of table: those that contain the data and those that contain the metadata. This second group of tables constitutes **the catalogue** of the database.

This characteristic of relational system implementations is known as **reflexivity**. A DBMS typically manages the catalogue by using structures similar to those in which the database instance is stored.



DATA MODIFICATION IN SQL

- Statements for:
  - insertion INSERT
  - deletion DELETE
  - change of attribute values UPDATE
- All the statements **can operate on a set of tuples** (set-oriented)
- In the condition it is possible to access other relations



INSERTIONS /1

```
INSERT INTO TableName [(AttributeList)] <VALUES(ListofValues)|SELECT SQL>
```

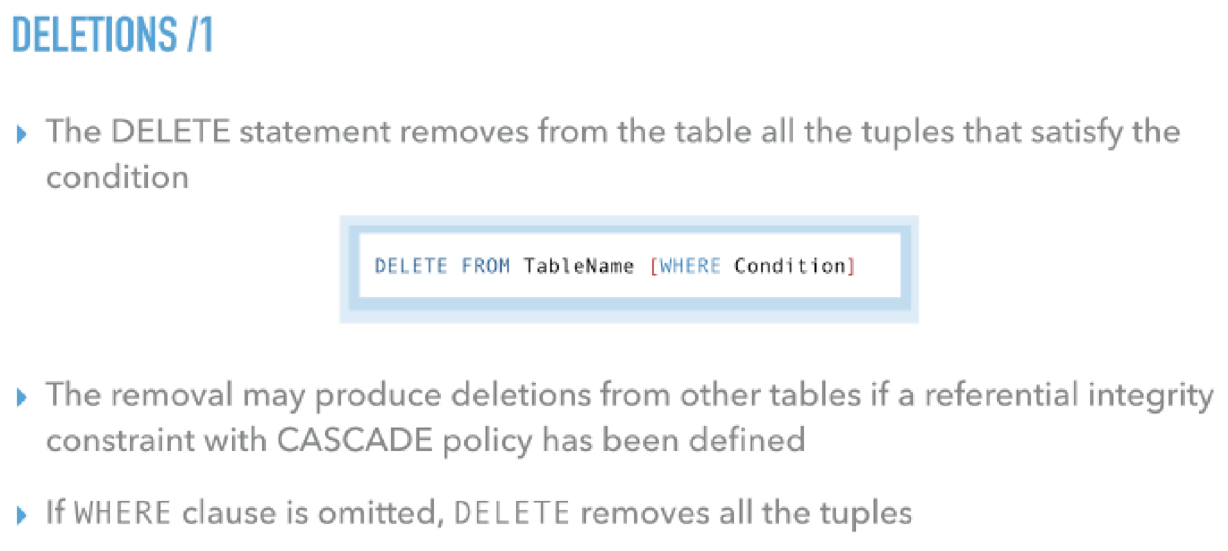
- Using Values

```
INSERT INTO Department(DeptName,City) VALUES ('Production','Toulouse')
```

- Using a subquery

```
INSERT INTO LondonProducts(
    SELECT Code, Description
    FROM Product
    WHERE ProdArea = `London`
)
```

- The ordering of the attributes (if present) and of values is meaningful (first value with the first attribute, and so on)
- If *AttributeList* is omitted, all the relation attributes are considered, in the order in which they appear in the table definition
- If *AttributeList* does not contain all the relation attributes, to the remaining attributes it is assigned:
  - the DEFAULT value (if defined)
  - the NULL value
  - PRIMARY KEYs might get special handling

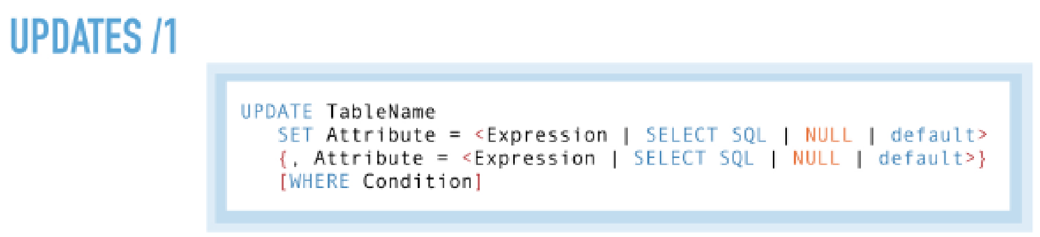


DELETIONS /1

- The DELETE statement removes from the table all the tuples that satisfy the condition

```
DELETE FROM TableName [WHERE Condition]
```

- The removal may produce deletions from other tables if a referential integrity constraint with CASCADE policy has been defined
- If WHERE clause is omitted, DELETE removes all the tuples



UPDATES /1

```
UPDATE TableName
   SET Attribute = <Expression | SELECT SQL | NULL | default>
   {, Attribute = <Expression | SELECT SQL | NULL | default>}
   [WHERE Condition]
```