



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének
együttes javítása a Pannon Egyetemen

DATABASE SYSTEMS THEORY AND PRACTICE

SZÉCHENYI 2020



MAGYARORSZÁG
KORMÁNYA

Európai Unió
Európai Szociális
Alap



BEFEKTETÉS A JÖVŐBE

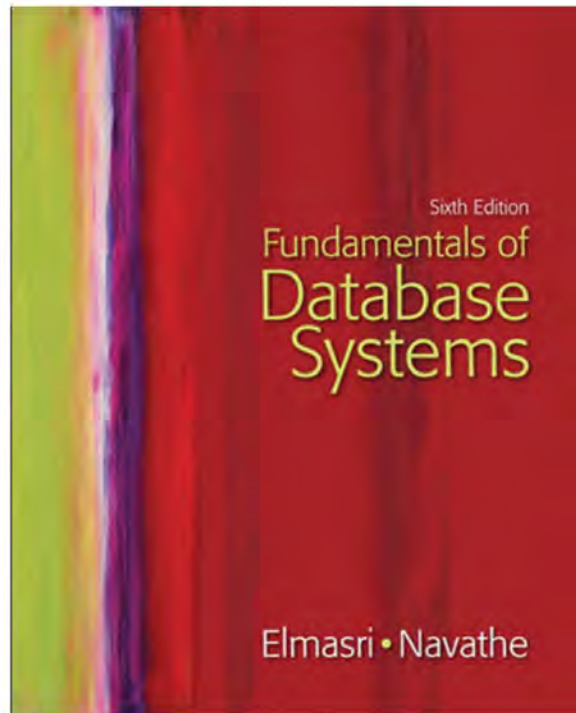
Slides was compiled by Ágnes Vathy-Fogarassy

- Introduction
- Data models
- The relational model
- Normalization
- Entity-Relationship model
- Extended Entity-Relationship model
- Mapping (E)ER model to relational model
- Big Data and NoSQL

Recommended literature

The slideshow is based on the book:

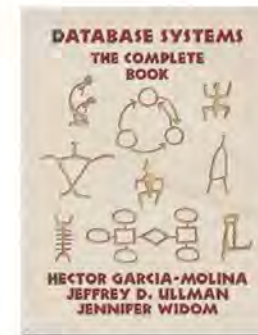
- Elmasri & Navathe: Fundamentals of Database Systems



Related Books

- Silberschatz & Korth & Sudarshan: Database System Concepts
- Thomas M. Connolly, Carolyn E. Begg:
Database Systems:
A Practical Approach to Design, Implementation, and Management
- Molina & Ullman & Widom:
Database Systems: The Complete Book
Equivalent with:

Ullman & Widom:
A First Course in Database Systems
+
Molina & Ullman & Widom:
Database System Implementation





Introduction

Table of Content

- Basic concepts
- DBMS functionalities
- Data abstraction
- Data independence
- Database users
- Two-tier and three-tier architectures
- Advantages of using the database approach
- Extending database capabilities

Basic Concepts

- **Data:**
 - Known facts that can be recorded and have an implicit meaning.
- **Database (DB):**
 - A collection of related data.
- **Database Management System (DBMS):**
 - A collection of programs that enables users to create and maintain a database.
- **Database System:**
 - The DBMS software together with the data itself. Sometimes, the applications are also included.

Data and Metadata

Database approach vs.
file processing approach (traditional programming)

Self-describing nature of database management systems:

- A DBMS **catalog** stores the description of a particular database.
- The description is called **metadata**, which includes e.g.:
 - data structures,
 - data types,
 - constraints,
 - ...
- This allows the DBMS software to work with different database applications.
- In traditional file processing approach programs are constrained to work with only one specific database.

Example of a Simple Database (Relational)

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

Example of a Simplified Database Catalog

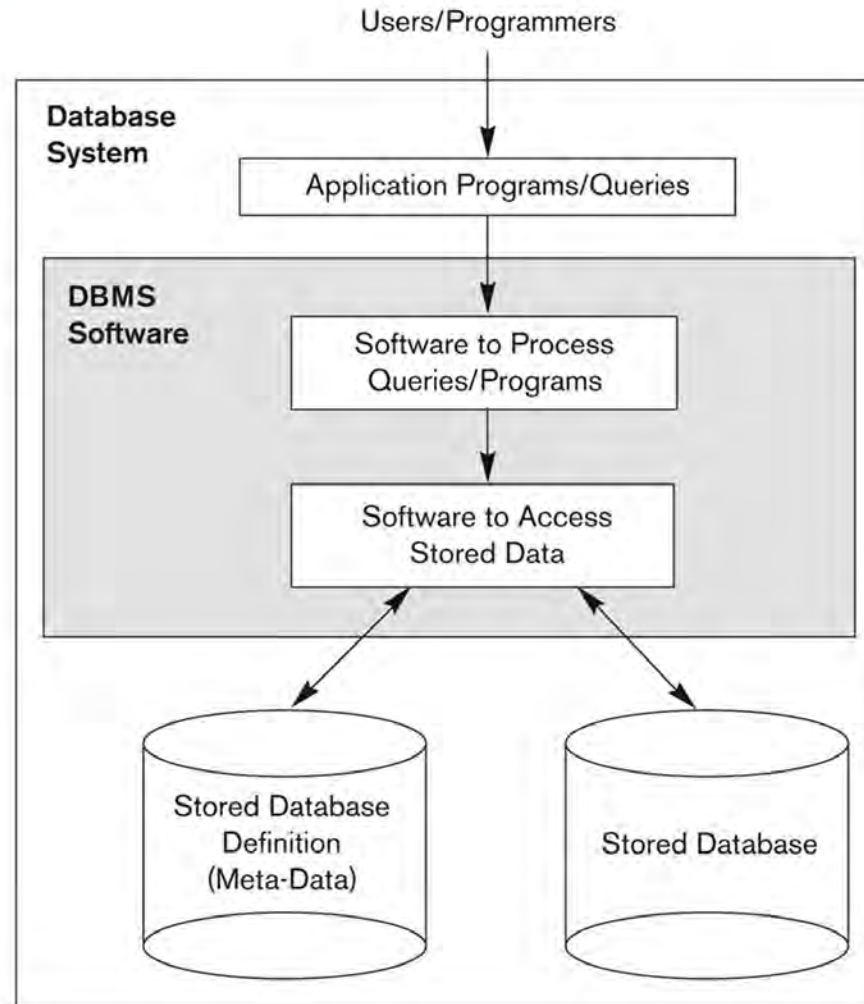
RELATIONS

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

Simplified Database System Environment



Typical DBMS Functionalities

- **Defining** a database: specifying *data types, structures, and constraints*
 - **Metadata**: descriptive information of the database
- **Constructing** the database: the process of storing the data on some storage medium controlled by the DBMS
- **Manipulating** the database:
 - **Retrieval**: querying data, generating reports
 - **Modification**: insertions, deletions and updates to its content
- **Sharing a database** and **transaction processing**:
Allows multiply users and programs to access the DB simultaneously – yet, keeping all data valid and consistent.

Typical DBMS Functionalities

- Other features:
 - **Protecting** *the database*:
 - *system protection* against hardware or software malfunction
 - *security protection* against unauthorized or malicious access
 - **Maintaining** *the database* and associated programs over the lifetime of the database application
 - **Presentation** and **visualization** of data

Data Abstraction

Data Abstraction

- A DBMS provides users with a conceptual representation of data that does not include many of the details.
 - e.g. how the data is stored, how the operations are implemented
- Levels of abstraction:
 - Physical (Internal) level
 - Logical (Conceptual) level
 - View (External) level

Levels of Data Abstraction

- **Physical level:** lowest level, describes *how the data are actually stored*.
 - **Physical schema:** the database design at the *physical* level
- **Logical level:** next-higher level, describes *what data* are stored in the database, and *what relationships* exist among those data.
 - **Logical schema:** the *logical structure* of the database
- **View level:** highest level, application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.
 - **External schema:** different *views* of the database

Physical Level

Physical level:

- physical representation of the database on the computer, describes how the data are actually stored
- covers the physical implementation of the database to achieve optimal runtime performance and storage space utilization
- **Physical schema:** the database design at the *physical* level
 - It includes e.g.:
 - storage space allocation for data and indexes
 - record description for storage
 - record placement
 - data compression and encryption
 - it may be managed by the operation system under the direction of the DBMS

Logical Level

Logical level:

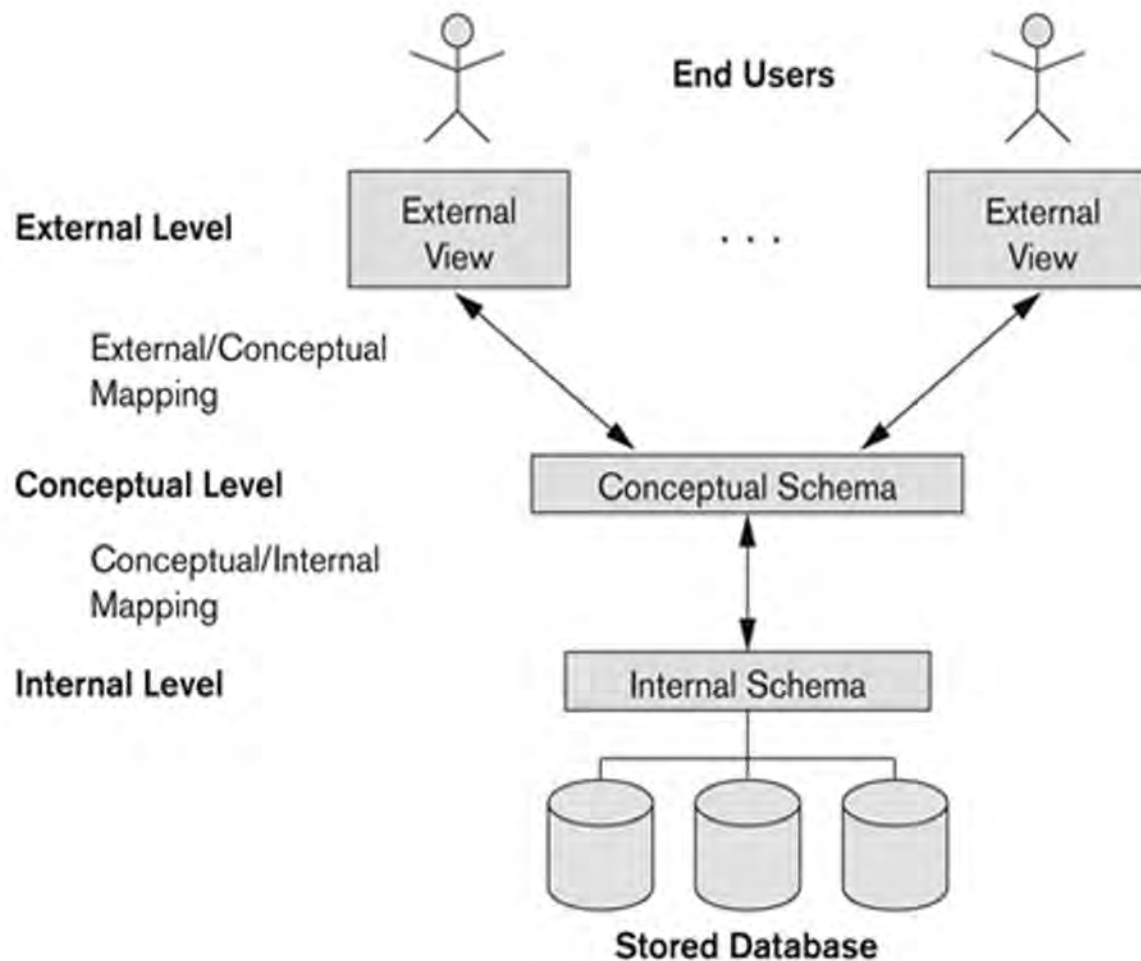
- next-higher level, describes *what data are stored* in the database, and *what relationships exist* among those data
- its a *complete view* of the data requirement of the organization that is *independent of any storage considerations*
- **Logical schema:** the logical structure of the entire database
 - The conceptual level represents:
 - entities, attributes, relationships
 - constraints on the data
 - semantic information about the data
 - security and integrity information
- the conceptual level supports each external view

View Level

View level:

- highest level, users' views of the database
- represents the application logic
- hides information for security purposes
- **External schema:** different views of the database
 - consists of a number of different external views of the database
 - describes that part of database that is relevant to a particular user
 - different views may have different representation of the same data

ANSI-SPARC Three-Level Architecture



Data Independence

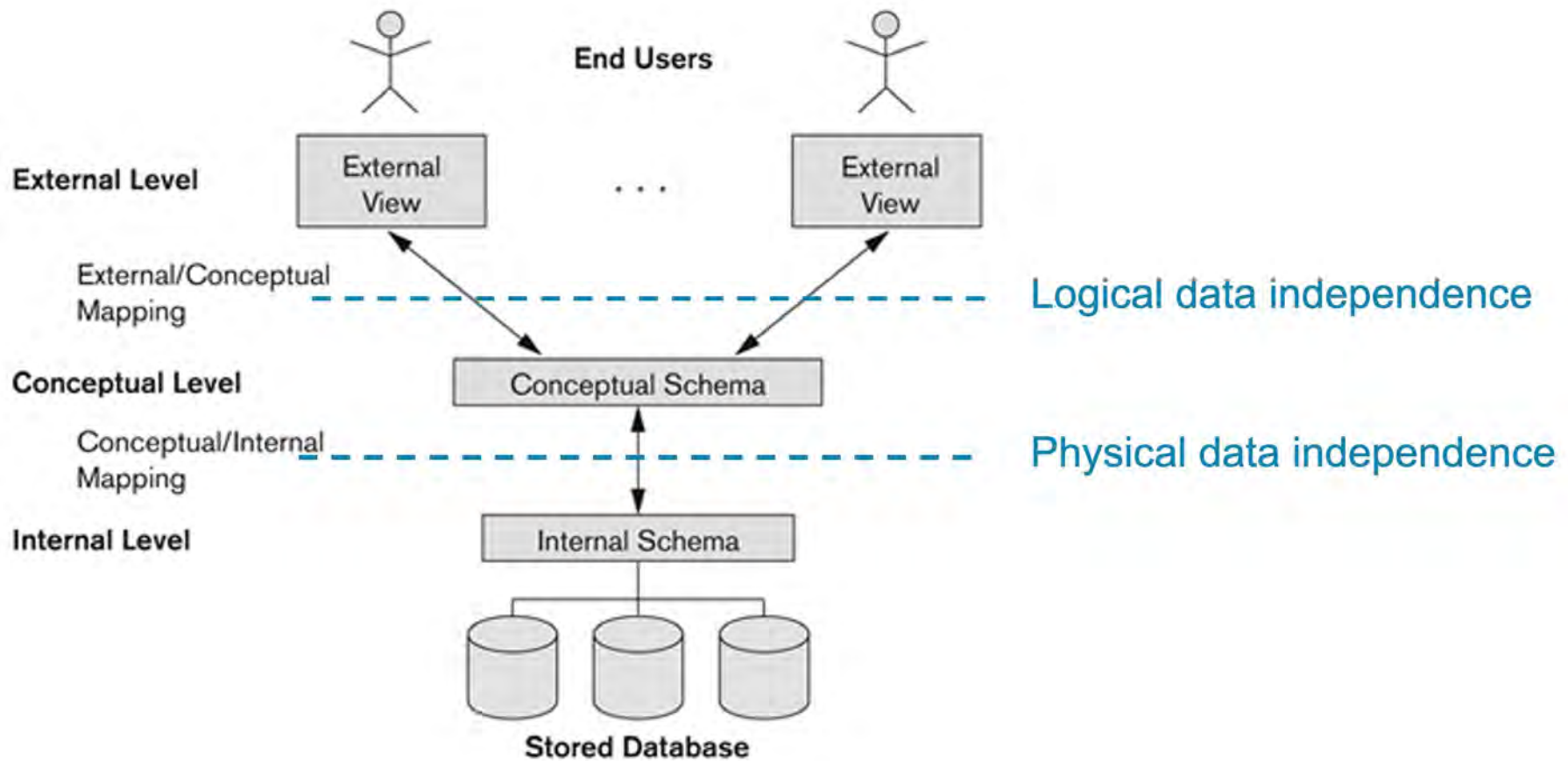
A major objective for the three-level architecture is to provide ***data independence***:

Each higher level of the data architecture is immune to changes of the next lower level of the architecture.

Two types of data independence:

- logical data independence
- physical data independence

Data Independence



Data Independence

- **Logical data independence:** The ability to change the logical (conceptual) schema without changing the external schema (user view).
 - For example: the addition or removal of new entities, attributes, or relationships to the conceptual schema should be possible without having to change existing external schemas or having to rewrite existing application programs.
- **Physical data independence:** The ability to change the physical schema without changing the logical schema.
 - For example: a change to the internal schema, such as using different file organization or storage structures, storage devices, or indexing strategy, should be possible without having to change the conceptual or external schemas.

Database Users

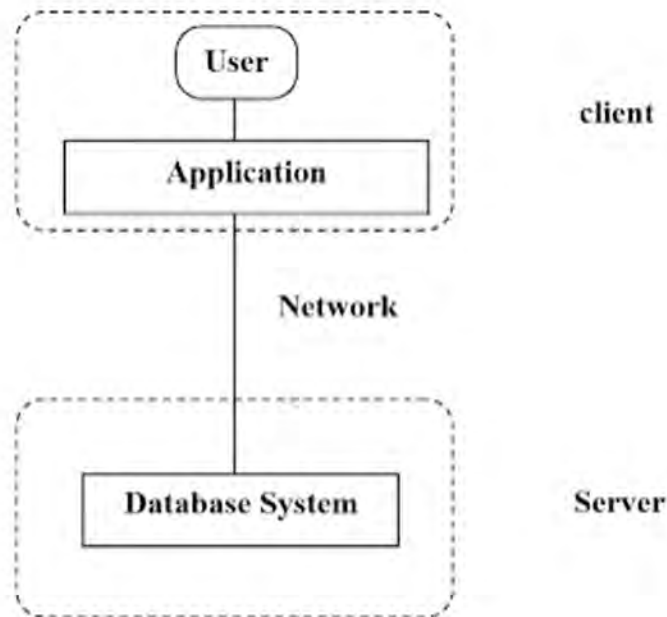
- **DBMS implementers:**
 - They build database management systems.
- **Database designers:**
 - Responsible to define the content, the structure, the constraints, and functions or transactions against the database. They must communicate with the end-users and understand their needs.
- **Application programmers:**
 - Computer professionals who write application programs.
- **End-users:**
 - **Causal users:** They have great knowledge of query languages, and they access data by entering different queries from the terminal end (e.g. **data analyst**).
 - **Naive users:** Their task is to just use the developed application and get the desired results. They do not have any knowledge about database systems (e.g. clerical staff).

Database Users

- Database administrator (DBA):
 - A person who has central control over the data and programs that access those data.
 - Responsible for:
 - authorizing access to the database
 - coordinating and monitoring its use
 - monitoring efficiency of operations
 - periodically backing up the database
 - ensuring that enough free disk space is available for normal operations

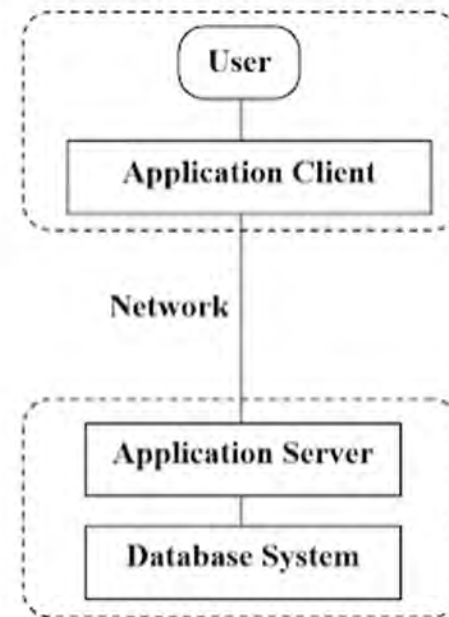
Two-tier Architecture vs. Three-tier Architecture

Two-tier architecture



- Client (tier 1) manages user interface and runs applications.
- Server (tier 2) holds database and DBMS.

Three-tier architecture



- Client program contains presentation logic only.

Where to Place Business Logic?

ORM: Object Relational Mapper

(e.g.: Entity Framework)

- **LINQ (Language Integrated Query):**
Accessing database without writing SQL code (parameterized queries)
- Data entities generated by the ORM are effectively business objects. Instead of working with datasets, these entities can enforce business rules that can interact with the presentation layer.

Stored procedures

- business logic is implemented in **stored procedures**
- the same procedures can be called from multiple, different frameworks and languages

BUT:

- good SQL knowledge can be hard to find (expensive)
- DBA will be required for performance tuning
- significant server-side resource requirements

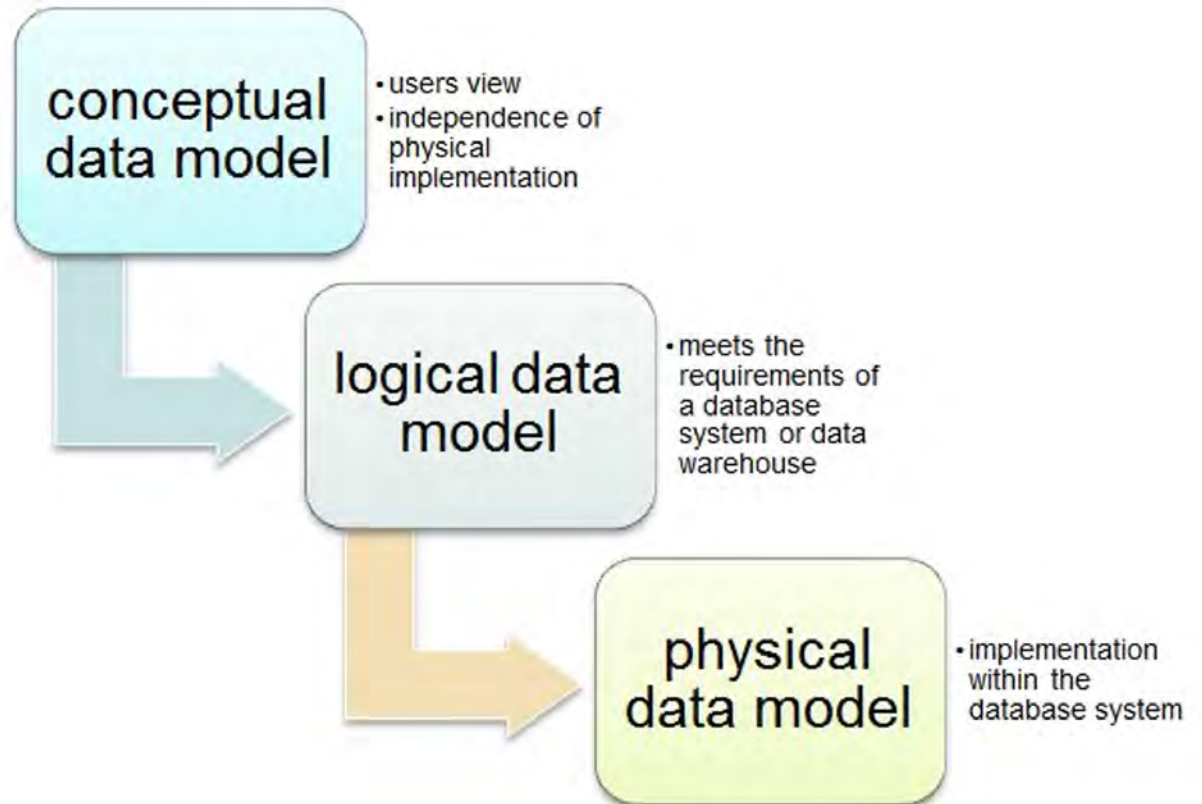
Advantages of Using the Database Approach

- logical and physical data independencies
- restricting unauthorized access to data
- providing persistent storage for program objects
- providing storage structures (e.g. indexes) for efficient query processing
- providing backup and recovery services
- providing multiple interfaces to different groups of users
- representing complex relationships among data
- enforcing integrity constraints on the database
- drawing inferences and actions from the stored data using deductive and active rules
 - active database systems, stored procedures, triggers,...

Extending Database Capabilities

Traditional applications:

- numeric and textual databases
- New functionalities are being added to DBMSs in the following areas:
 - image storage and management
 - audio and video data management
 - data warehousing
 - data mining
 - spatial data management
 - time series and historical data management
 - handling XML, and JSON data
 - graph databases
 - filetable solutions
 - ...



<http://www.wikixbrl.info>

Data Models

Table of Content

- Types of data models
- Data models
- Designing data models
- Schemas and instances

Why Data Modelling?

- **Data modeling** in software engineering is the *process of creating a data model* for an information system by applying *formal data modeling techniques*.
- The way data is stored is very important for the way in which it can be accessed and manipulated.
- A good data model is one for which:
 - It is easy to write correct and understandable queries.
 - No change is needed for minor changes in the problem domain.
 - If the problem domain changes significantly, it is easy to modify the data model and associated programs.

Two Broad Types of Database Activities

Data model – for what?

- OLTP- Online Transaction Processing
 - short transactions
 - simple queries
 - touch small portions of data
 - frequent updates
- OLAP- Online Analytical Processing
 - long transactions
 - complex queries
 - touch large portions of data
 - infrequent updates
- Big Data

Data Models

- **Data model** is a collection of tools for describing:
 - data,
 - data relationships,
 - data semantics,
 - data constraints.
- Main categories of data models:
 - Conceptual (high-level) data models
 - Logical (implementation-based) data models
 - Physical (low-level) data models
- Semi-structured data model (XML, JSON)

Conceptual Data Models

- Define *what* the system contains, *without any implementation aspects*.
 - Describe data at the conceptual and view level.
- Provide concepts that are close to the way many users perceive data.
- Provide fairly flexible structuring capabilities.
- Different models, for example:
 - Entity-Relationship model (ER model)
 - Extended Entity-Relationship model (EER model)
 - Data Structure Diagram (DSD)

Logical Data Models

- Define *how* the system should be implemented regardless of the DBMS.
- Unlike conceptual models, they are used to
 - specify overall *logical structure* of the database, and
 - provide a *higher-level description of the implementation*.
- They provide concepts that may be understood by the end users but that are not too far removed from the way data is organized within the computer.
- The most widely-accepted models (for different purposes):
 - Relational data model
 - Snowflake schema, star schema
 - Object-oriented model
 - Object-relational model
 - Network model
 - Hierarchical model

Physical Data Models

- Describe the details of *how data is stored* in the computer (e.g.: indexes, data types, files, constraints)
- Generally meant for computer specialist, not for typical end users.

The screenshot shows the 'Staff : Table' design view in Microsoft Access. The table has five fields: StaffID, FirstName, SurName, ScalePoint, and DOB. The 'StaffID' field is selected, and its properties are shown in the 'Field Properties' pane. The 'General' tab is active, showing the field size as 'Integer', format as 'Auto', decimal places as 'Auto', input mask as empty, caption as empty, default value as empty, validation rule as '>=1001 And <=1199', validation text as 'Must be >=1001 And <=1199', required as 'No', and indexed as 'Yes (No Duplicates)'. A note on the right states: 'A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.'

Field Name	Data Type	Description
StaffID	Number	StaffIdentifiers (1001 to 1199)
FirstName	Text	PersonNames (String <=12 Characters)
SurName	Text	PersonNames (String <=12 Characters)
ScalePoint	Number	ScalePoints (Number, from 1 to 8)
DOB	Date/Time	StaffBirthDates (Date, >= 21 years before today)

Field Properties

General | Lookup

Field Size: Integer

Format: Auto

Decimal Places: Auto

Input Mask:

Caption:

Default Value:

Validation Rule: >=1001 And <=1199

Validation Text: Must be >=1001 And <=1199

Required: No

Indexed: Yes (No Duplicates)

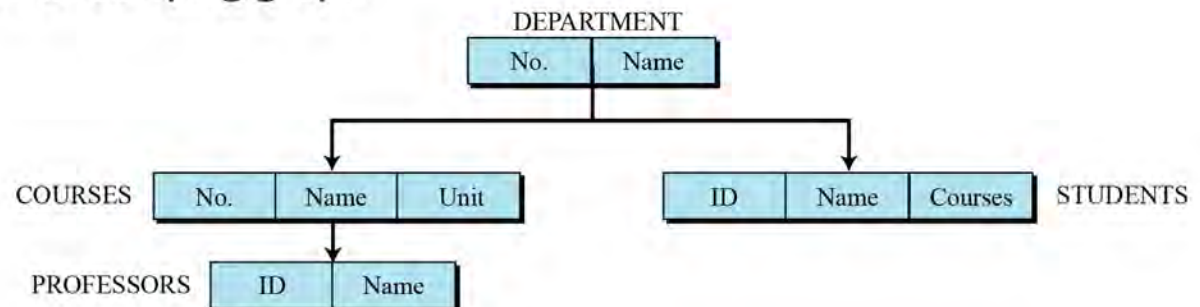
A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

Data Models

WELL-DEFINED (STRUCTURED)

The Hierarchical Model

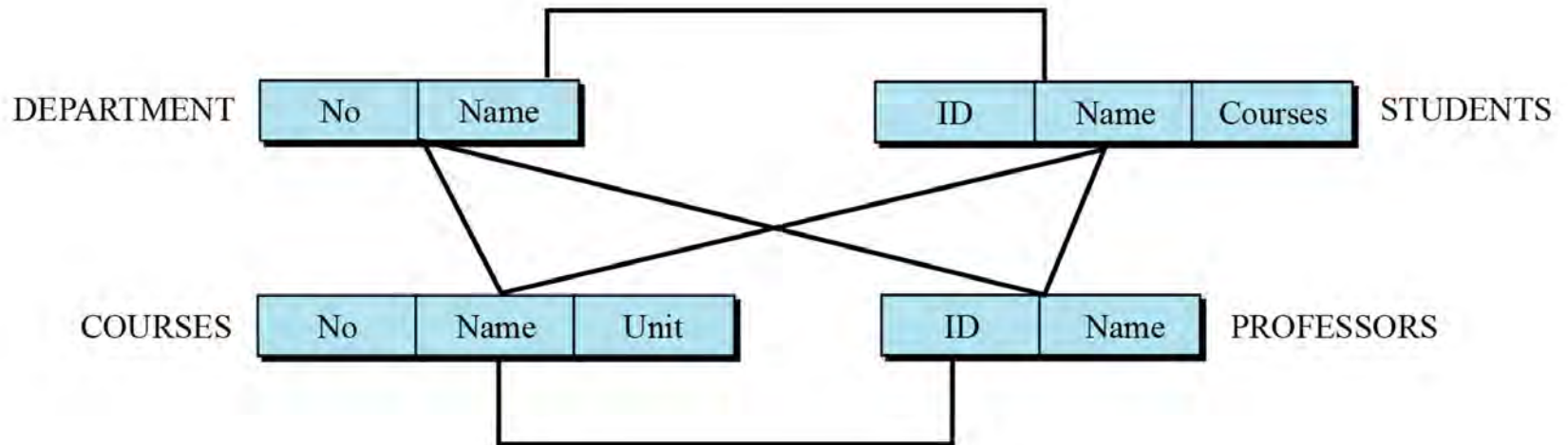
- Organization of the records is a collection of **trees**.
 - information represented as parent/child relationships
 - each parent can have many children
 - each child has only one parent
 - relationships formed in the graph must be such that only one-to-many or one-to-one relationships exist between a parent and a child
 - no cycles in the underlying graph



- *We are no longer using it.*

The Network Model

- The network model allows each record to have multiple parent and child records, forming a generalized **graph structure**.



- We are no longer using it.*

The Relational Model

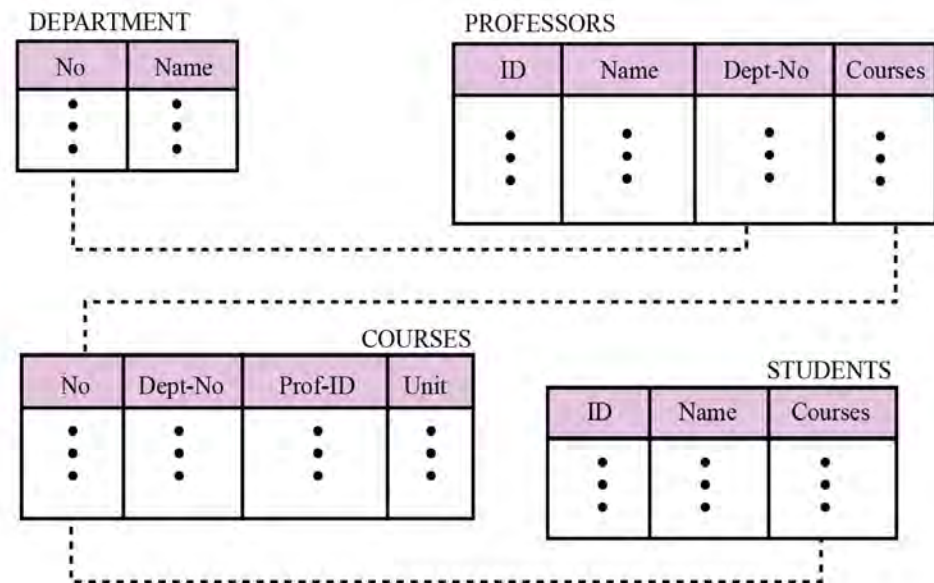
Dr. E. F. Codd published his influential paper:

[“A Relational Model of Data for Large Shared Data Banks”](#)

In *Communications of the ACM* **13** (6): 377–387 (1970)



- First database model to be described in formal mathematical terms.
- Based on the concept of mathematical *relations*.
 - Data and relationships are represented by a collection of *tables*.
- Each table has a number of columns with unique names.
- SQL: Query with high-level language: simple yet expressive.
- Several commercial and open source products are based on the relational model.

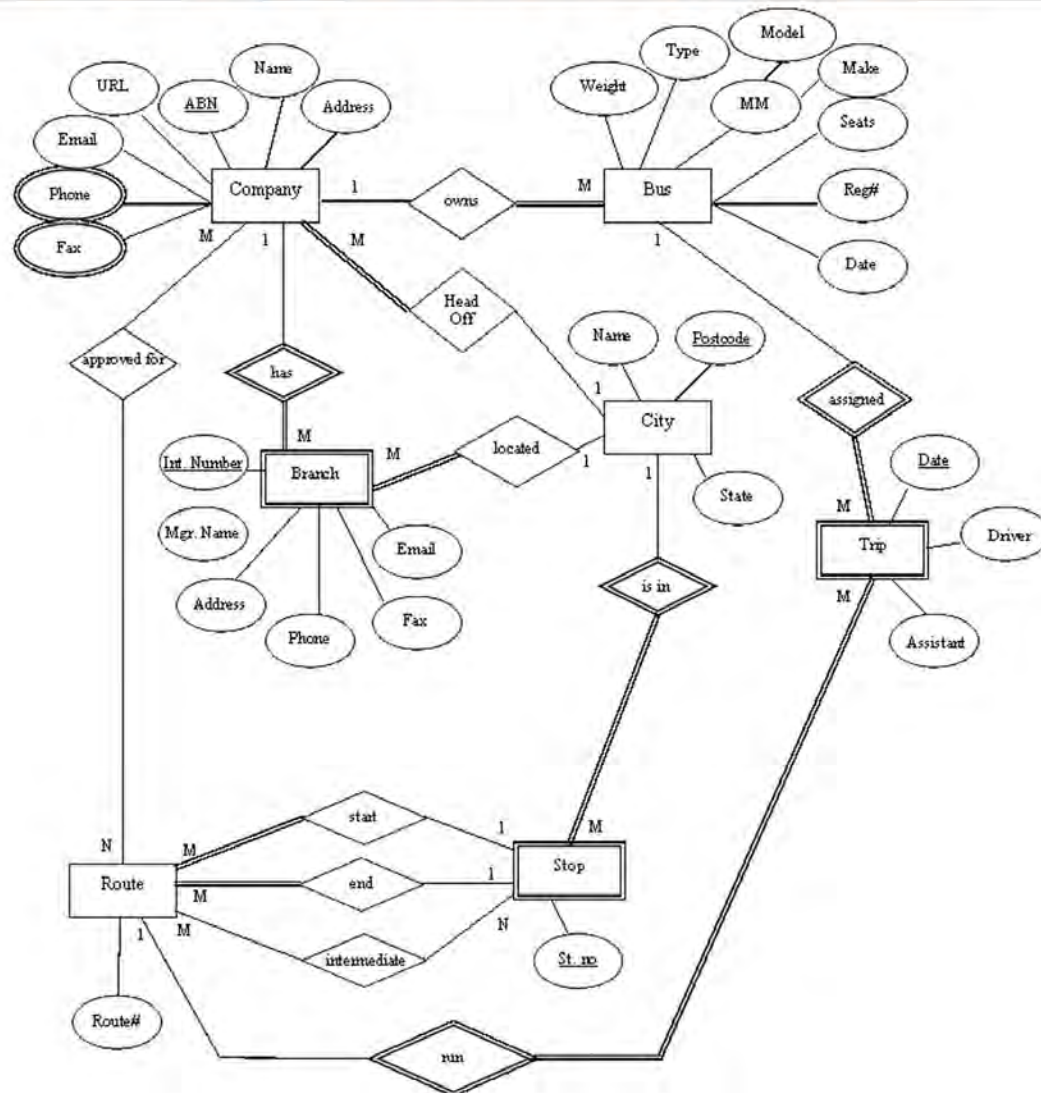


The Entity-Relationship Model



- Proposed by Peter Chen [1976].
- A *graphical approach* to database design.
- The Entity-Relationship model is based on a perception of the world as consisting of a collection of **objects (entities)** and **relationships** among these object.
- It uses concepts such as:
 - **Entity**: a real-world object or concept
(e.g. employee, project, book, course, exam)
 - **Attribute**: property of interest that further describes an entity
(e.g. employee's name, employee's salary, book's title)
 - **Relationship**: represents an association among two or more entities
(e.g. employee *works on* project)

Example of Entity-Relationship Model



The Object-Oriented Model

- A core **object-oriented data model** consists of the following basic object-oriented concepts:
 - **object and object identifier**: Any real world entity is uniformly modelled as an object, and associated with a unique id.
 - **attributes and methods**:
 - every object has:
 - a *state*: the set of values for the attributes of the object, and
 - a *behaviour*: the set of methods which operate on the state of the object
 - *sending a message*: The only way in which one object can access the data of another object is by invoking the method of that other object.
 - **encapsulation**: Internal parts of the object, the instance variables and the method code, are not visible externally.
 - An attribute is an instance variable, whose domain may be any class: user-defined or primitive. An object contains values stored in **instance variables** within the object. Unlike the record-oriented models, these values are themselves objects. Thus objects contain objects to an arbitrarily deep level of nesting.

The Object-Oriented Model (cont.)

- A core **object-oriented data model** consists of the following basic object-oriented concepts:
 - **class**: a group of the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class (instance-of relationship).
 - A class may also be primitive (no attributes), e.g., integer, string, Boolean.
 - **Class hierarchy and inheritance**: derive a new class (subclass) from an existing class (superclass).
 - The subclass inherits all the attributes and methods of the existing class and may have additional attributes and methods.
 - Single inheritance (class hierarchy) vs. multiple inheritance (class lattice).

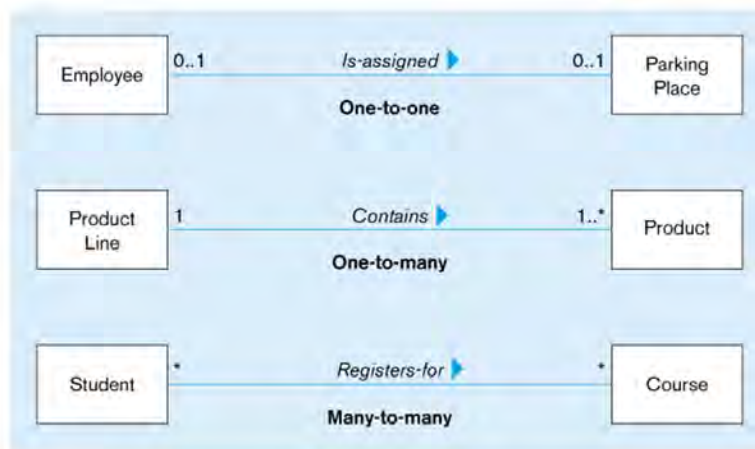
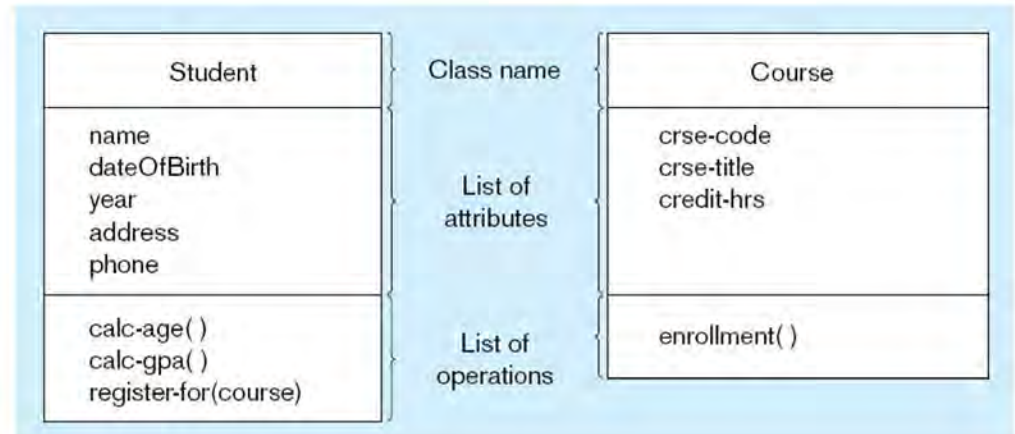
The Object-Oriented Model (cont.)

- Unlike entities in the ER model, each object has its own unique identifier, independent of the values it contains.
 - Two object containing the same values are distinct.
 - Distinction is created and maintained in physical level by assigning distinct object identifiers.

Object-oriented modelling

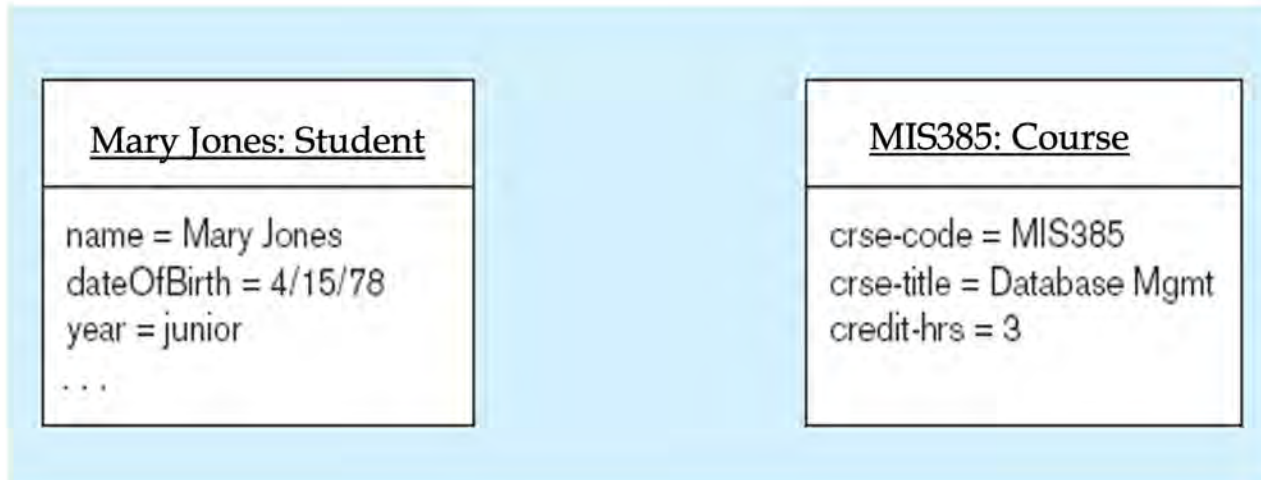
- **UML Class diagram:** shows the static structure of an object-oriented model:

- object classes,
- internal structure,
- relationships



Object-oriented modelling

- **UML Object diagram:** shows instances that are compatible with a given class diagram.



Object-Relational Data Model – Motivation

- Relational model: clean and simple
 - great for administrative data
 - not as good for other kinds of data (e.g. multimedia, networks, CAD)
- Object-Oriented models: complicated, but some influential ideas
 - complex data types
 - object identity/references
 - ADTs (encapsulation, behavior goes with data)
 - inheritance
- Idea: build DBMS based on OO model

Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added *data types* (e.g. row type).
- Preserve relational foundations, in particular the declarative access to data, while extending modelling power.
- Allow attributes of tuples to have complex types, including non-atomic values such as *nested relations*.
- *Inheritance*: type and table hierarchies
- Provide upward compatibility with existing relational languages.
- An object-relational DBMS (ORDBMS) supports an extended form of SQL called SQL3.
- Especially good for special application areas: e.g. text and document databases, geographic data, biomedical data.

Data Models

SEMI STRUCTURED

XML: Extensible Markup Language

- Originally intended as a document markup language not a database language.
- The ability to specify new tags, and to create nested tag structures made XML a great way to exchange **data**, not just documents.
- A wide variety of tools is available for parsing, browsing and querying XML documents/data.

```
<?xml version="1.0" ?>
<!-- Bookstore with no DTD -->
- <Bookstore>
-   <Book ISBN="ISBN-0-13-713526-2" Price="85" Edition="3rd">
      <Title>A First Course in Database Systems</Title>
      <Authors>
        <Author>
          <First_Name>Jeffrey</First_Name>
          <Last_Name>Ullman</Last_Name>
        </Author>
        <Author>
          <First_Name>Jennifer</First_Name>
          <Last_Name>Widom</Last_Name>
        </Author>
      </Authors>
    </Book>
-   <Book ISBN="ISBN-0-13-815504-6" Price="100">
      <Remark>Buy this book bundled with "A First Course" -- a great deal!</Remark>
      <Title>Database Systems: The Complete Book</Title>
```


JSON: JavaScript Object Notation

- standard for serializing data objects, usually in files
- human-readable, useful for data interchange
- also useful for representing and storing semi structured data
- parsers for many languages

- No longer tied to JavaScript

```
{ "Books":  
  [  
    { "ISBN":"ISBN-0-13-713526-2",  
      "Price":85,  
      "Edition":3,  
      "Title":"A First Course in Database Systems",  
      "Authors":[ {"First_Name":"Jeffrey", "Last_Name":"Ullman"},  
                   {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }  
    ,  
    { "ISBN":"ISBN-0-13-815504-6",  
      "Price":100,  
      "Remark":"Buy this book bundled with 'A First Course' - a great deal!",  
      "Title":"Database Systems:The Complete Book",  
      "Authors":[ {"First_Name":"Hector", "Last_Name":"Garcia-Molina"},  
                   {"First_Name":"Jeffrey", "Last_Name":"Ullman"},  
                   {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }  
  ]  
}
```

Designing Database Systems

Designing Database Systems

Two main activities:

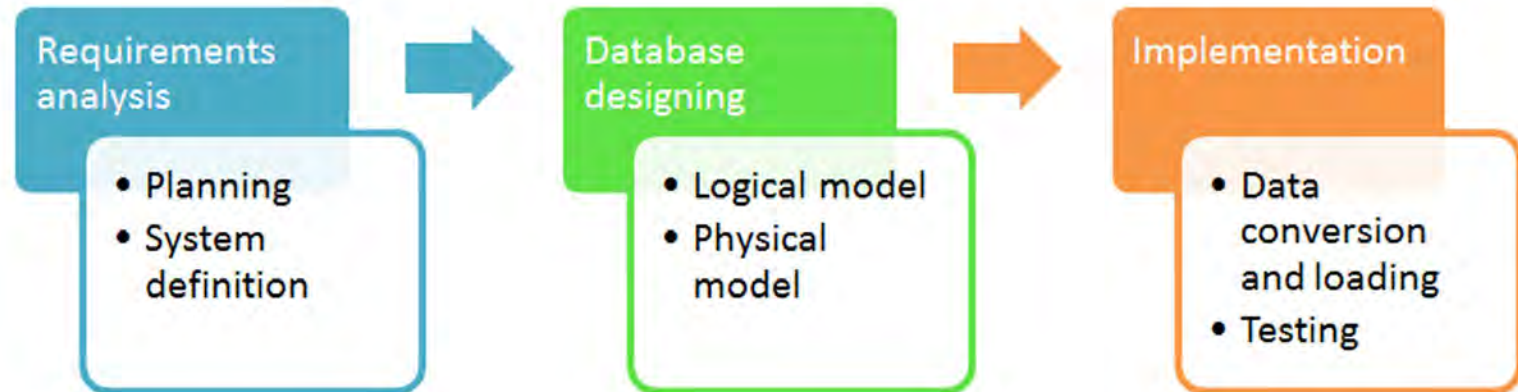
- Database design

- Normalization
- Conceptual database design ((E)ER – modelling)

- Applications design

- Applications design focuses on the programs and interfaces that access the database.
- Generally considered as a part of software engineering.

The Database Development Process



The Database Development Process

- **Requirements analysis**

- **Planning:** Planning of entire Database Development Life Cycle.
- **System definition:** Defining the scope and boundaries of the proposed database system. Collecting of **data** and **functional requirements**.

- **Database designing**

- **Logical model:** Developing a database model based on requirements, without any physical implementations or specific DBMS considerations.
- **Physical model:** This stage implements the logical model of the database taking into account the DBMS and physical implementation factors.

- **Implementation**

- **Data conversion and loading:** Importing and converting data from the old system into the new database, or load data into the system.
- **Testing:** Identification of errors in the newly implemented system. It checks the database against the requirement specifications.

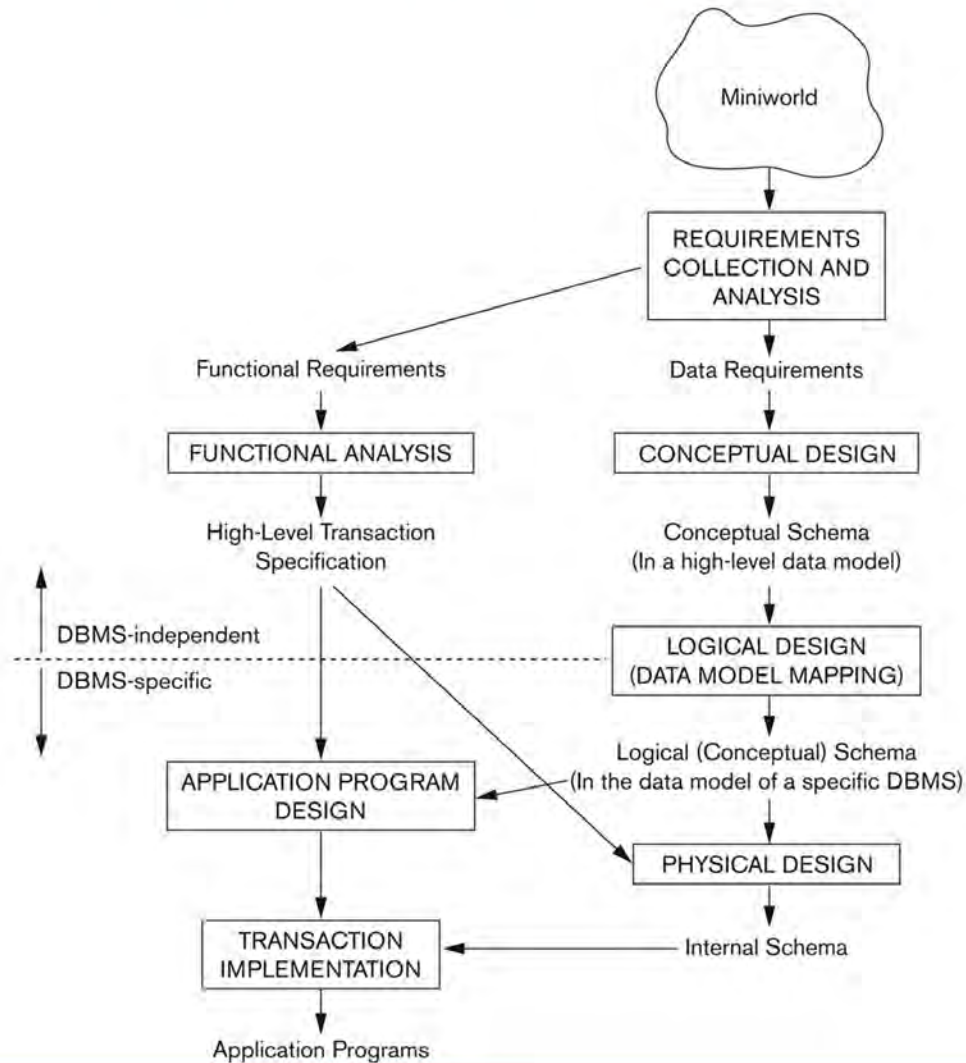
Database Design Process with Normalization

- Requirement collection and analysis
- *Normalization:*
 - A database design technique which *organizes tables* in such a manner that it *reduces redundancy* and *dependency of data*.
 - It divides larger tables to smaller tables and link them using relationships.
- Physical DB design
- Implementation

The Conceptual Database Design Process

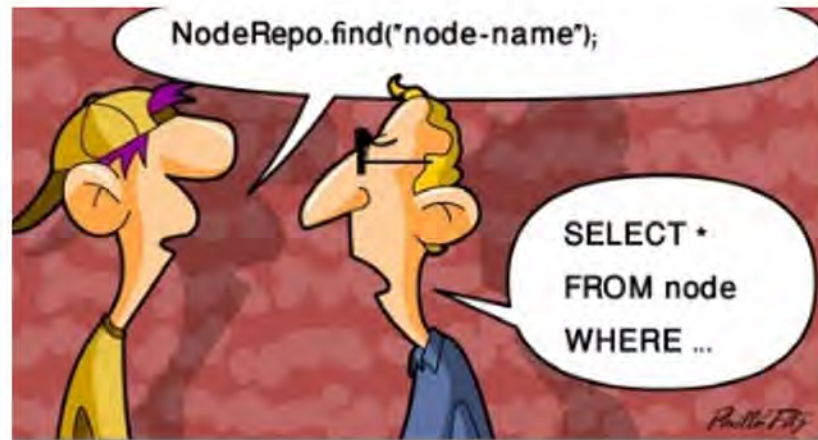
- Requirement collection and analysis
- *Conceptual DB design using a high-level model*
 - Designing a conceptual database model (e.g. ER model).
- *Logical DB design (data model mapping)*
 - Transforming the high-level conceptual schema into an implementation data model (e.g. relational model).
- Physical DB design
- Implementation

Overview of Database Design Processes



Object-Relational Mapping

- Object-relational mapping (ORM):
 - NOT a database design technique!
 - a *programming technique* for converting data between incompatible type systems in object-oriented programming languages
 - Object-relational mapping converts the complex object values into groups of simpler values for storage in the database.



OO does not talk to SQL

Schemas and Instances

Schemas and Instances

- **Schema**: the structure of the database
 - It is specified during the database design process.
 - Not expected to change frequently.
- **Instance**: the actual content of the database at a particular point in time
 - The actual data in database may change frequently.
 - Many database instances can correspond to the same database schema.

Example of a Database Schema

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
----------------	--------------------	-------

Example of a Database Instance

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

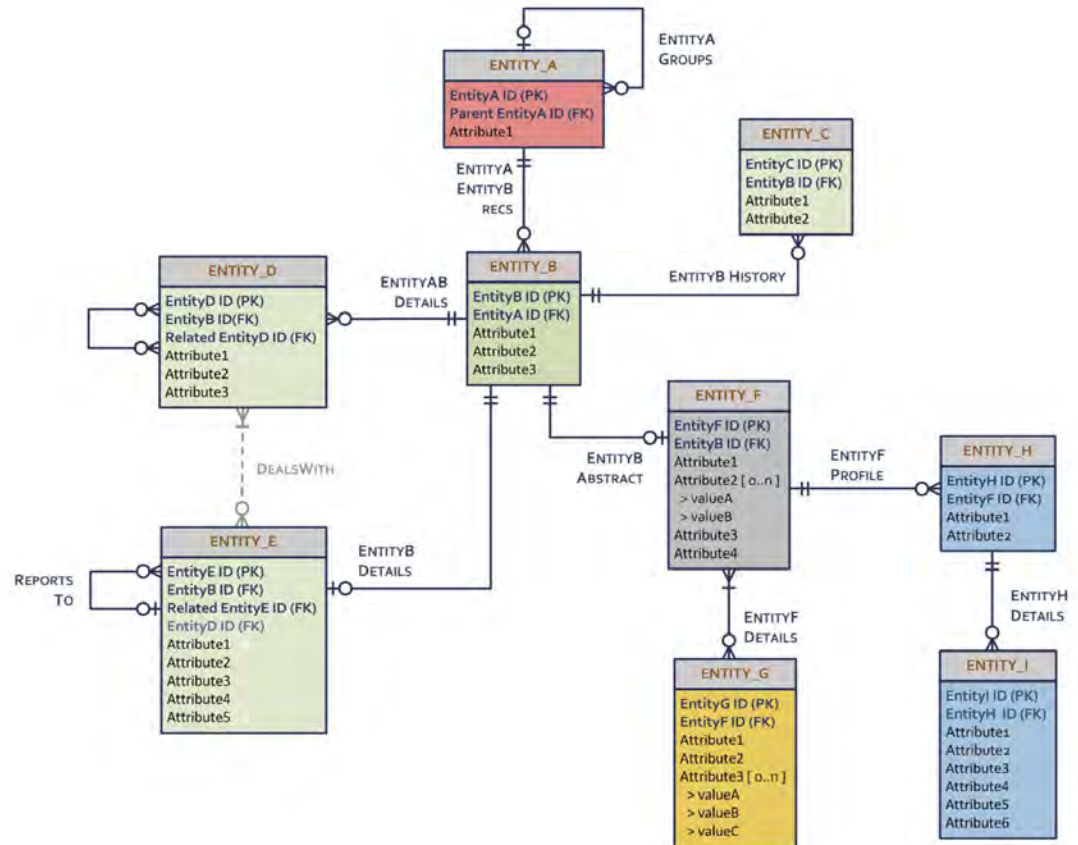
Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310



The Relational Model

Table of Content

- Relational model concepts
- Formal definitions
- Keys
- Relational model constraints
- Transactions

Basic Concepts

- The relational model is based on the mathematical concept of the **relation**.
- A **relation is represented** as a **two-dimensional table** with *columns* and *rows*.
- Each **table** has a unique name (in the database).
- **Tuple, record = Row**: a collection of related data values.
 - Each row has a value of *an item* or *set of items* that uniquely identifies that row in the table.
 - Sometimes *row-ids* or *sequential numbers* are assigned to identify the rows in the table.
- **Attribute = Column**: called by its *column name* or *attribute name*.
- **Relational Database Management System (RDBMS)**: DBMS based on the relational model.

Relation – Example

Table name

Attributes (columns)

BRANCH

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Tuples (rows, records)

The diagram illustrates a database relation named 'BRANCH'. It consists of a table with four columns: 'branchNo', 'street', 'city', and 'postcode'. The table contains five rows of data. Labels with arrows identify the components: 'Table name' points to 'BRANCH', 'Attributes (columns)' points to the column headers, and 'Tuples (rows, records)' points to the data rows.

Domain

- Every attribute in a relation is defined on a domain.
- **Domain**: the set of allowable values for one attribute
 - consists of atomic values

Attribute	Domain Name	Meaning	Domain Definition
branchNo	BranchNumbers	The set of all possible branch numbers	character: size 4, range B001–B999
street	StreetNames	The set of all street names in Britain	character: size 25
city	CityNames	The set of all city names in Britain	character: size 15
postcode	Postcodes	The set of all postcodes in Britain	character: size 8
sex	Sex	The sex of a person	character: size 1, value M or F
DOB	DatesOfBirth	Possible values of staff birth dates	date, range from 1-Jan-20, format dd-mmm-yy
salary	Salaries	Possible values of staff salaries	monetary: 7 digits, range 6000.00–40000.00

Degree and Cardinality

- **Degree**: number of attributes
- **Cardinality**: number of tuples
 - changes as tuples are added or deleted

BRANCH

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Degree

Cardinality

Formal Definition – Cartesian Product

- Consider two sets, D_1, D_2 .
- **Cartesian product** $D_1 \times D_2$ is a set of *all ordered pairs*, where first element is member of D_1 , and second element is member D_2 .
- Example: $D_1 = \{2, 4\}$, and $D_2 = \{1, 3, 5\}$

$$D_1 \times D_2 = \{(2,1), (2,3), (2,5), (4,1), (4,3), (4,5)\}$$

Formal Definition – Relation

Given domains: $D_1, D_2, \dots, D_n$

A **relation** r is a mathematical relation of degree n on the domains $D_1, D_2, \dots, D_n$ which is a subset of the *Cartesian product* of the domains that define r :

$$r \subseteq D_1 \times D_2 \times \dots \times D_n$$

A relation in a relational data model is *a set of n -tuples* $(a_1, a_2, \dots, a_n)$, where each $a_i \in D_i$.

Formal Definition – Schemas

- A **relation schema** R , denoted by $R(A_1, A_2, \dots, A_n)$, is made up of a relation name R and a list of attributes $A_1, A_2, \dots, A_n$.
 - It is used to describe a relation.
 - Example:

STUDENT(NeptunID, Name, BirthDate, Address)

- **Relational database schema**: A set of relation schemas, each with a distinct name.

Properties of Relations

A relation has the following properties:

- *Relation name* is distinct from all other relation names in relational database schema.
- *Each cell of relation contains exactly one atomic (single) value* (or it is empty).
- *Each attribute in the relation has a distinct name.*
- *Values of an attribute are all from the same domain.*
- *Each tuple is distinct*, there are no duplicate tuples.
- Order of attributes has no significance.
- Tuples can appear in any order.
- A special **NULL** value is used to represent values that are unknown.

Keys

- **Superkey (SK):**

- A set of attributes (SK) of R such that no two tuples in any valid relation instance $r(R)$ will have the same value for SK . That is, for any distinct tuples t_1 and t_2 in $r(R)$, $t_1[SK] \neq t_2[SK]$.
- A superkey may contain extraneous attributes.

- **Key or Candidate key (K):**

- A candidate key is a **minimal superkey**. That is, removal of any attribute from a candidate key (K) results in a set of attributes that is not a superkey.

- **Primary key (PK):**

- If a relation has several **candidate keys**, one is chosen arbitrarily to be the primary key.
- Notation: The primary key attributes are underlined.

Keys (cont.)

- **Alternate key:** The candidate key that are not selected to be the primary key.

Simple vs. Composite keys:

- **Composite key:** When a key consists of more than one attribute we call it composite key.
- **Simple key:** When a key consists of exactly one attribute we call it simple key.

Keys – Example

Example: The STUDENT relation schema:

STUDENT(*NeptunID*, *PersonalID*, *Name*, *Phone*, *BirthDate*, *Home*)

Superkeys:

- SK1 = {NeptunID}
 - SK2 = {PersonalID, Name}
 - SK3 = {Name, Home, BirthDate}
-
- {PersonalID, Name} is a superkey but *not* a candidate key.
 - SK3 = {Name, Home, BirthDate} is a composite key.
 - If SK1 = {NeptunID} is chosen to be the primary key, the SK3 = {Name, Home, BirthDate} is an composite alternate key.

Keys – Example

- The CAR relation with two candidate keys:
 - *LicenseNumber* and *EngineSerialNumber*
- *LicenseNumber* was chosen to be the *primary key*.
- *EngineSerialNumber* is an alternate key.

CAR	<u>LicenseNumber</u>	EngineSerialNumber	Make	Model	Year
	Texas ABC-739	A69352	Ford	Mustang	96
	Florida TVP-347	B43696	Oldsmobile	Cutlass	99
	New York MPO-22	X83554	Oldsmobile	Delta	95
	California 432-TFY	C43742	Mercedes	190-D	93
	California RSK-629	Y82935	Toyota	Camry	98
	Texas RSK-629	U028365	Jaguar	XJS	98

Characteristics of Primary Key

- A relation always has a primary key.
- The values of the *primary key* are unique.
- The *primary key attributes* of relational schema R can not have NULL values in any tuple of $r(R)$.
 - This is because primary key values are used to *identify* the individual tuples.
 - Other attributes of R may be similarly constrained to disallow NULL values, even though they are not members of the primary key.
- **composite primary key**: primary key involving more than one attributes

Foreign Key

Foreign key: A relation R_1 may include among its attributes the primary key of another relation (say R_2). This attribute (or set of attributes) is called a *foreign key from R_1 , referencing R_2* .

- R_1 : referencing relation
- R_2 : referenced relation
- A row t_1 in R_1 is said to **reference** a tuple t_2 in R_2 if $t_1[FK] = t_2[PK]$.
- The *foreign key reference* can be displayed in a relational database schema as a *directed arc* from $R_1.FK$ to R_2 .
- **composite foreign key:** foreign key involving more than one attributes

Characteristics of Foreign Key

- The **value in the foreign key column** (or columns) of the *referencing relation* R_1 can be either:
 - a value of an existing primary key value of the corresponding primary key PK in the *referenced relation* R_2 , or
 - a NULL value
 - In this case, the FK in R_1 should *not* be a part of its own primary key.
- If the *referenced primary key* is composite, the *referencing foreign key* must be **composite**, as well!

Foreign Key – Example

BRANCH



<u>branchNo</u>	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

STAFF

<u>staffNo</u>	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

In the BRANCH relation
BranchNo is the **primary key**,
and in the STAFF relation it is a
foreign key.

Referencing relation: STAFF

Referenced relation: BRANCH

Constraints

- **Constraints** are *conditions* that *must hold on all valid relation instances*.
 - **Invalid state**: a database state that does not obey all integrity constraints
- There are three main types of constraints:
 - Domain constraints
 - Entity constraints
 - Referential integrity constraints
- User defined constraints
 - E.g. an upper limit of 20 has been placed upon the number of staff that may work at a branch office.

Domain and Entity Constraints

- **Domain constraints:**

- every element from a relation should respect the *type* and *restrictions* of its corresponding attribute
- **NOT NULL constraint:** the value of the attribute can not be the NULL value
- E.g.:
 - *age* is integer (type constraint)
 - *age* ≥ 18 (restriction – CHECK constraint)
 - *name* is NOT NULL (NOT NULL constraint)

- **Entity constraints:**

- **PRIMARY KEY constraint:** every table must have a primary key and the column or columns chosen to be the *primary key* should be *unique* and *not null*.
- **UNIQUE constraint:** values of the attribute have to be unique

Referential Integrity Constraint

- **Referential integrity constraint:**

- It is specified between two relations.
- Used to maintain the consistency among tuple's in the two relations.
- A tuple in one relation that refers to another relation *must refer to an existing tuple in the referencing relation* (or it can be NULL value).

Referential Integrity – Example

Consider the situation where we have two tables:

EMPLOYEE and DEPARTMENT.

- The EMPLOYEE table has a foreign key attribute entitled *Dno*, which refers to the department in the DEPARTMENT table for which the employee works.

EMPLOYEE (SSN, FName, Minit, Lname, Bdate, Address, Sex, Salary, *Dno*)

DEPARTMENT (Dnumber, Dname)



Constraints and Data Manipulations on Relations

Data manipulations:

- **INSERT** a tuple
- **DELETE** a tuple
- **MODIFY** a tuple
- Constraints should not be violated by *any* operations!
- Several data manipulations may have to be grouped together.
 - Data manipulations may *propagate* to cause other updates automatically.

Maintenance of Constraints

- Violation of *domain* or *entity constraints*: REJECTION
- Violation of the *integrity constraint*:
 - Trigger additional updates:
 - CASCADE option,
 - SET NULL option,
 - SET DEFAULT option,
 - NO ACTION option (check after the manipulation of data),
 - RESTRICT option (check before trying to execute the statement)
 - Execute a user-specified error-correction routine.

Cascading Techniques

Cascading delete:

- Whenever rows in the *referenced table* are *deleted*, the respective rows of the *referencing table* with a *matching foreign key column* will get *deleted* as well.

Cascading update:

- Whenever the value of a *primary key in the referenced table* is *updated*, the respective rows of the *referencing table* with a matching foreign key will get *updated* as well.

Referential Integrity and Cascading - Example

Referential integrity enforces the following rule:

- The value of *Dno* in any tuple of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT, or the value of *Dno* can be NULL if the employee does not belong to a department.

EMPLOYEE (SSN, FName, Minit, Lname, Bdate, Address, Sex, Salary, *Dno*)

DEPARTMENT (Dnumber, Dname)



Optional rules may be:

- If the primary key for a record in the DEPARTMENT table *changes*, all corresponding records in the EMPLOYEE table can be modified using a *cascading update*.
- If a record in the DEPARTMENT table is *deleted*, all corresponding records in the EMPLOYEE table can be deleted using a *cascading delete*.

Example – Company

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	<u>Relationship</u>
-------------	-----------------------	-----	-------	---------------------

Example – Company

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

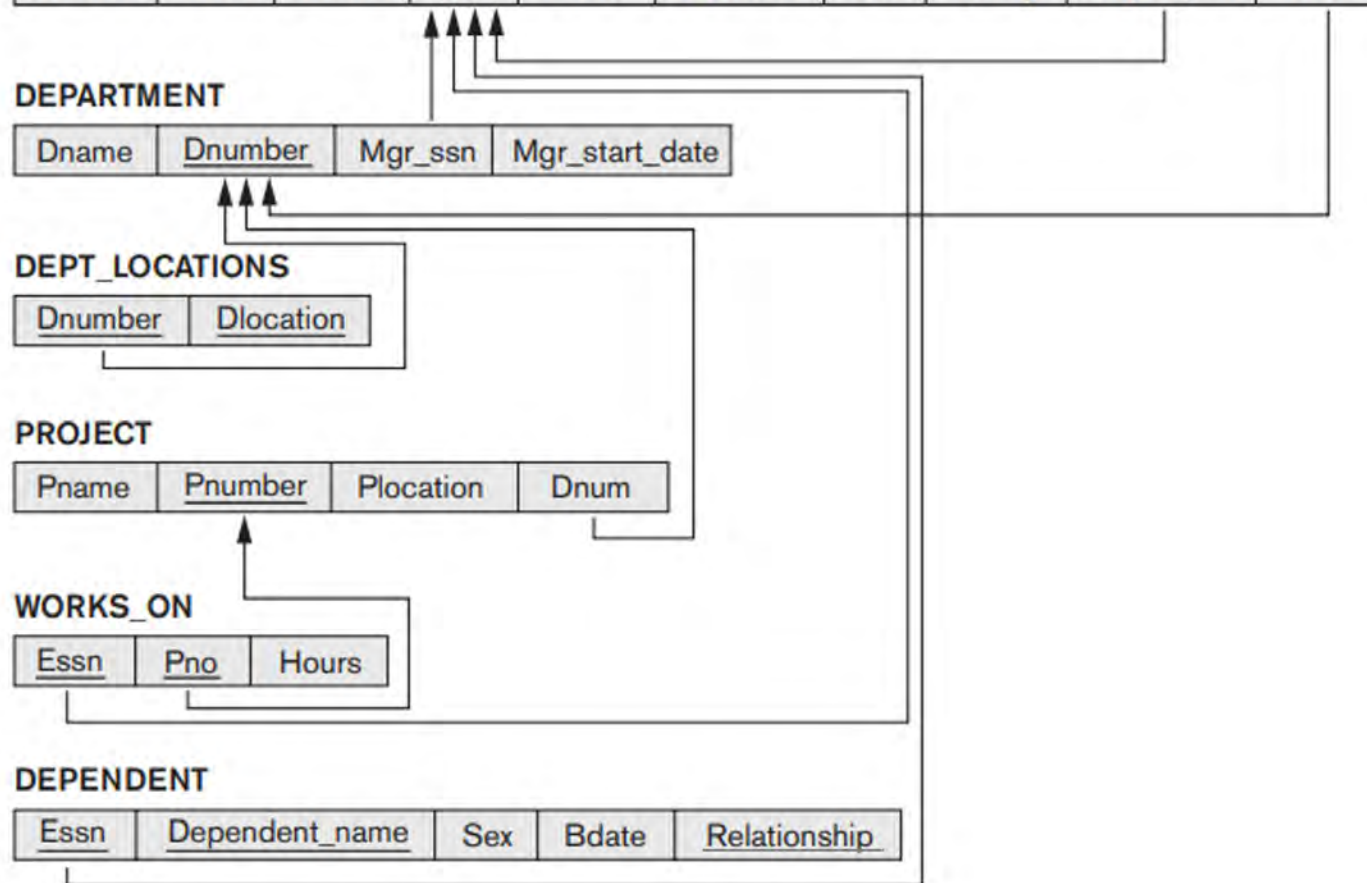
Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	<u>Relationship</u>
-------------	-----------------------	-----	-------	---------------------



Transactions

Transaction: a *computation unit* that accesses and possibly modifies a database

- Generally, it is a group of tasks, which may contain several low-level tasks.
 - In special cases, a low-level task may also be considered as a single transaction.
- Before and after running of the transaction, the database must be consistent.
- During the transaction execution the database may be inconsistent.
- Purposes of transaction management:
 - deal with failures of various kinds (e.g. hardware failures, system crashes)
 - handle concurrent user requests: concurrent execution of multiple transactions

Transaction Support

OLTP (Online Transaction Processing):

allows hundreds of concurrent transactions to execute per second simultaneously.

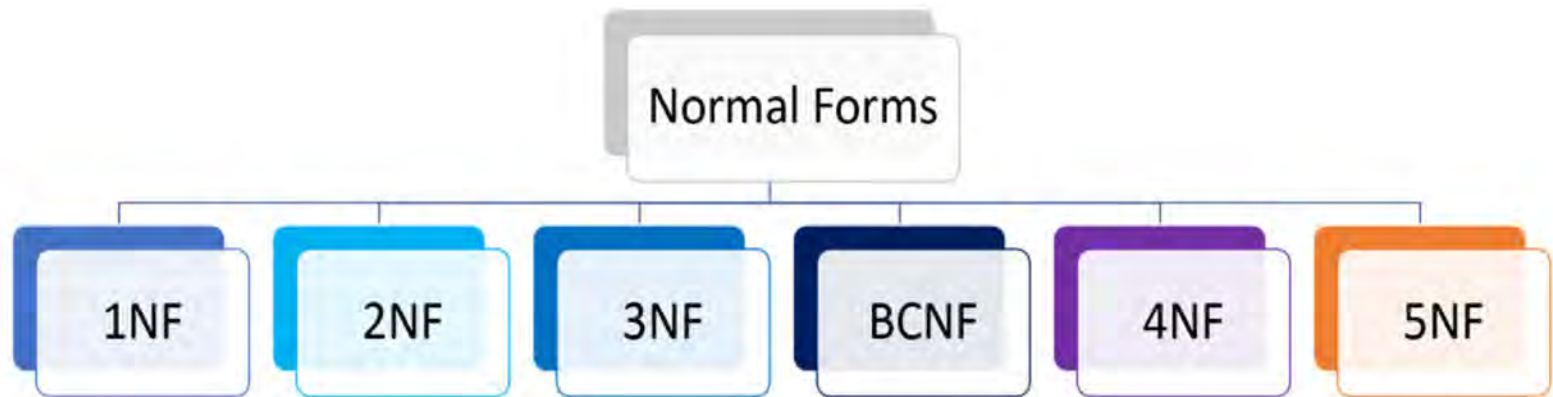
- DBMS supports a set of **concurrent users** to retrieve from and to update the database.
- **Concurrency control** within the DBMS guarantees that **each transaction** is correctly executed or aborted.
- **Recovery subsystem** ensures each completed transaction has its effect permanently recorded in the database.

ACID Properties of Transactions

A transaction in a database system must maintain the ACID properties.

ACID

- **Atomicity**: each transaction is atomic: do all or nothing
- **Consistency**: any transaction will bring the database from one valid state to another
- **Isolation**: concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially
- **Durability**: once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors



Normalization

Table of Content

- Redundancy and anomalies
- Functional dependencies (FDs)
- Normalization of relations and different Normal Forms
- Example (CarInsurance)

Redundant Information in Tuples and Anomalies

- **Data redundancy**: unnecessary duplication of data
- Redundant information causes:
 - storage wastage
 - **Anomalies**:
 - Insertion anomalies
 - Deletion anomalies
 - Modification anomalies
- **Database consistency**: the database contains only valid data
 - Data redundancy may result in **data inconsistency**.
 - Anomalies leads to an **inconsistent database state**.

Anomalies

- **Insert anomaly:**

- occurs when certain attributes cannot be inserted into the database without the presence of other attributes

- **Update anomaly:**

- exists when one or more instances of duplicated data is updated, but not all

- **Delete anomaly:**

- exists when certain attributes are lost because of the deletion of other attributes (rows)

Redundancy – Example I.

					Redundancy	
EMP_DEPT						
Ename	<u>Ssn</u>	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

Redundancy – Example II.

EMP_PROJ			Redundancy	Redundancy	
<u>Ssn</u>	<u>Pnumber</u>	Hours	Ename	Pname	Plocation
123456789	1	32.5	Smith, John B.	ProductX	Bellaire
123456789	2	7.5	Smith, John B.	ProductY	Sugarland
666884444	3	40.0	Narayan, Ramesh K.	ProductZ	Houston
453453453	1	20.0	English, Joyce A.	ProductX	Bellaire
453453453	2	20.0	English, Joyce A.	ProductY	Sugarland
333445555	2	10.0	Wong, Franklin T.	ProductY	Sugarland
333445555	3	10.0	Wong, Franklin T.	ProductZ	Houston
333445555	10	10.0	Wong, Franklin T.	Computerization	Stafford
333445555	20	10.0	Wong, Franklin T.	Reorganization	Houston
999887777	30	30.0	Zelaya, Alicia J.	Newbenefits	Stafford
999887777	10	10.0	Zelaya, Alicia J.	Computerization	Stafford
987987987	10	35.0	Jabbar, Ahmad V.	Computerization	Stafford
987987987	30	5.0	Jabbar, Ahmad V.	Newbenefits	Stafford
987654321	30	20.0	Wallace, Jennifer S.	Newbenefits	Stafford
987654321	20	15.0	Wallace, Jennifer S.	Reorganization	Houston
888665555	20	Null	Borg, James E.	Reorganization	Houston

Examples of Update Anomalies

Problems of **update anomaly**:

- Example I:

In EMP_DEPT, if we change the value of one of the attributes of particular department – e.g. the manager of a department – we must update the tuples of all employees who work in that department. Otherwise the database will become inconsistent.

- Example II:

Consider the relation EMP_PROJ. Changing the name of project number 1 from “ProductX” to “ProductA” may cause this update to be made for all employees working on project 1. If we do not do so, the database will contain inconsistent (conflicting) data.

Example of an Insert and Delete Anomalies

Consider the relation:

EMP_PROJ(Ssn, Pnumber, Hours, Ename, Pname, Plocation)

- The problem of **insert anomaly**:
 - We can not insert a project unless an employee is assigned to it.
 - Conversely, we can not insert an employee unless he/she is assigned to a project.
- The problem of **delete anomaly**:
 - When a project is deleted, it will result in deleting all the employees who work on that project.
 - Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

Functional dependencies

Functional Dependency

Definition:

A **functional dependency (FD)**, denoted by $X \rightarrow Y$, is a relationship between two set of attributes (X and Y) in the following way:

- For any relation R , attribute Y is functionally dependent on attribute X (usually the PK), if for every valid instance of X , that value of X uniquely determines the value of Y .
- That is, for any two tuples t_1 and t_2 in any relation instance $r(R)$:
If $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$.

Notes:

- If $X \rightarrow Y$ true in R , this does not say whether or not $Y \rightarrow X$ is true in R .
- If K is a key of R , then K functionally determines all attributes in R .
 - since we never have two distinct tuples with $t_1[K] = t_2[K]$

Functional Dependency

$X \rightarrow Y$, we say:

- Y is **functionally dependent** on X
- X component of a tuple **functionally determine** the values of the Y component
- X called: **left-hand side** of the FD
- Y called: **right-hand side** of the FD

Examples of FD Constraints

- Social security number determines the first name of the employee:

$Ssn \rightarrow Fname$

- Project number determines project name and location:

$Pnumber \rightarrow Pname, Plocation$

- Employee Ssn and project number determines the hours per week that the employee works on the project:

$\{Essn, Pno\} \rightarrow Hours$

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

Inference Rules for FDs

Given a set of FDs F , we can *infer* additional FDs that hold whenever the FDs in F hold.

- Armstrong's inference rules:
 - **Reflexive**: If $Y \subseteq X$, then $X \rightarrow Y$
 - **Augmentation**: If $X \rightarrow Y$, then $XZ \rightarrow YZ$
 - Notation XZ stands for $X \cup Z$
 - **Transitive**: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- Some additional inference rules that are useful:
 - **Decomposition**: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
 - **Union**: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
 - **Pseudotransitivity**: If $X \rightarrow Y$ and $WY \rightarrow Z$, then $WX \rightarrow Z$

Normalization

Normalization of Relations

Normalization:

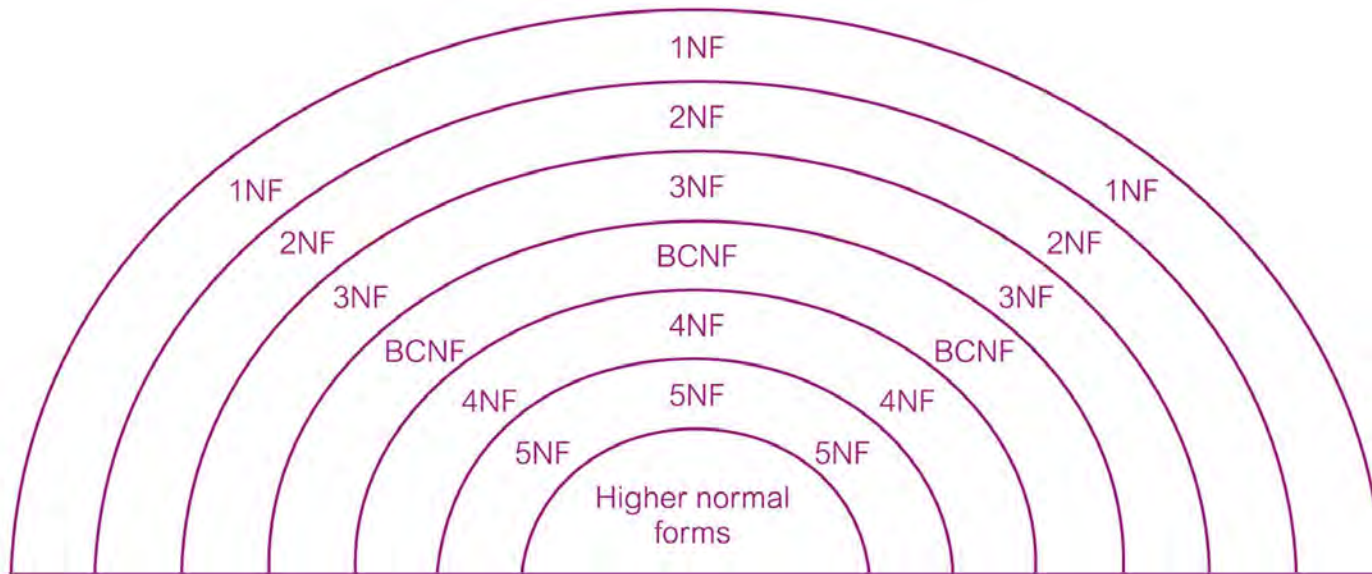
- Normalization is a relational database design process.
- It is a **top-down decomposition process**, where unsatisfactory relation schemas that do not meet certain conditions (normal form tests) are decomposed into smaller relation schemas that meet the tests.
- For the purpose of normalization we are interested in **identifying functional dependencies** between attributes **that holds for all time** (valid for any possible relation state).
- Often executed as a series of steps. Each step corresponds to a specific normal form, which has known properties.

Normal Forms (NFs)

Normal form:

- Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form.
- The **normal form of a relation** is the *highest NF* condition that it meets, and hence indicates the degree to which it has been normalized.
- **UNF**: A table that contains one or more repeating groups.
- **1NF**: based on the formal definition of the relation
- **2NF, 3NF, BCNF**: based on keys and FDs of a relation schema
- **4NF**: based on keys, multi-valued dependencies (MVDs)
- **5NF**: based on keys, join dependencies (JDs)

Normal Forms



Relationship between the normal forms.

Figure shows that relations in 2NF are also in 1NF and relations in 3NF are also in 2NF, and so on.

Unnormalized Normal Form (UNF)

Before normalization:

- Begin with a list of all of the fields that must appear in the database. Think of this as one big table.

Unnormalized form (UNF): A table that contains one or more repeating groups.

- **Repeating group:** an attribute or group of attributes within a table that occurs with multiple values for a single occurrence of the nominated key attributes of that table.

First Normal Form (1NF)

1 NF: Considered to be part of the definition of relation

1 NF disallows:

- composite attributes
- multivalued attributes
- nested relations: attributes whose values for an *individual tuple* are non-atomic

Normalization into 1NF

UNF:

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

We assume, that departments can have a number of locations.

1NF:

- No multivalued column, and
- PK is also determined

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	<u>Dlocation</u>
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Second Normal Form (2NF)

Definitions:

- **Prime attribute:** An attribute that is member of the primary key K .
- **Full functional dependency:** a FD $X \rightarrow Y$, where removal of any attribute from X means the FD does not hold any more.
- **Partial functional dependency:** A functional dependency $X \rightarrow Y$ is a partial dependency if some attribute $A \subset X$ can be removed from X and the dependency still holds. That is, for some $A \subset X$, $(X - \{A\}) \rightarrow Y$ still holds.

Examples:

- $\{Essn, Pno\} \rightarrow Hours$ is a full FD since neither $Essn \rightarrow Hours$ nor $Pno \rightarrow Hours$ hold
- $\{Ssn, Pnumber\} \rightarrow Fname$ is a partial dependency (not a full dependency), since $Ssn \rightarrow Fname$ also holds

Second Normal Form (2NF)

Definition:

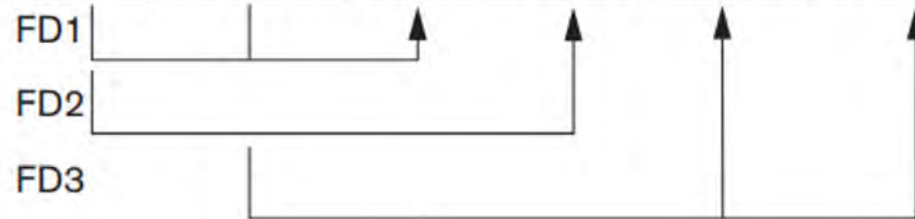
A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is *fully functionally dependent* on the primary key of R .

R can be *decomposed* into 2NF relations via the process of 2NF normalization.

Normalizing into 2NF

EMP_PROJ

<u>Ssn</u>	<u>Pnumber</u>	Hours	Ename	Pname	Plocation
------------	----------------	-------	-------	-------	-----------



2NF Normalization

EP1

<u>Ssn</u>	<u>Pnumber</u>	Hours
------------	----------------	-------



EP2

<u>Ssn</u>	Ename
------------	-------



EP3

<u>Pnumber</u>	Pname	Plocation
----------------	-------	-----------



Third Normal Form (3NF)

Definition:

- **Transitive functional dependency:** $X \rightarrow Y$ in a relation schema R is a transitive dependency if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R , and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

Examples:

- $Ssn \rightarrow Mgr_ssn$ is a **transitive FD**
 - since $Ssn \rightarrow Dnumber$ and $Dnumber \rightarrow Mgr_ssn$ hold
- $Ssn \rightarrow Fname$ is **non-transitive FD**
 - since there is no set of attributes X where $Ssn \rightarrow X$ and $X \rightarrow Fname$ hold

Third Normal Form (3NF)

Definition:

A relation schema R is in **third normal form (3NF)** if it is in 2NF *and* no non-prime attribute A in R is transitively dependent on the primary key.

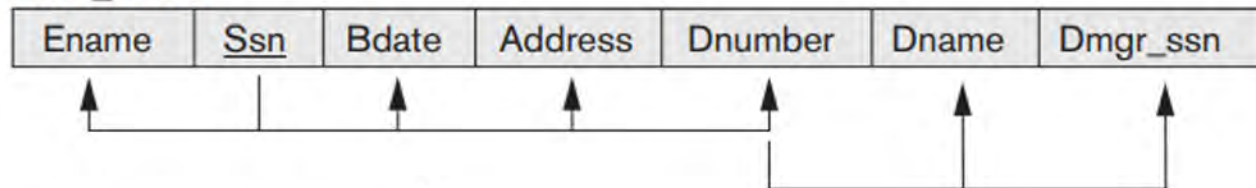
R can be *decomposed* into 3NF relations via the process of 3NF normalization.

- Notes:

- In $X \rightarrow Y$ and $Y \rightarrow Z$, with X as the primary key, we consider this a problem only if Y is not a candidate key.
 - E.g.: Consider $EMP(Ssn, Emp\#, Salary)$. Here, $Ssn \rightarrow Emp\# \rightarrow Salary$ and $Emp\#$ is a candidate key. As $Emp\#$ is a candidate key, the FD $Ssn \rightarrow Salary$ can not be considered as transitive FD, and therefore the EMP relation is in 3NF.

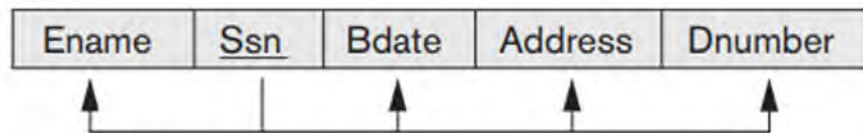
Normalizing into 3NF

EMP_DEPT

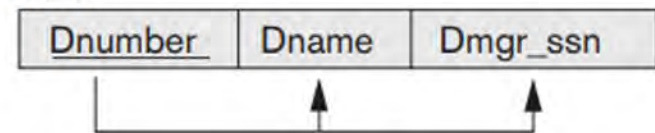


3NF Normalization

ED1



ED2



General Definitions

- **First normal form (1NF)**
 - Tables are said to be in first normal form when:
 - The table has a primary key.
 - No single attribute (column) has multiple values.
 - The non-key attributes (columns) depend on the primary key.
- **Second normal form (2NF)**
 - A relation that is in first normal form and every non-primary-key attribute is fully functionally dependent on *any candidate key*.
- **Third normal form (3NF)**
 - A relation that is in first and second normal form and in which no non-primary-key attribute is transitively dependent on *any candidate key*.

Boyce–Codd Normal Form (BCNF)

Definition:

A relation schema R is in **Boyce-Codd Normal Form (BCNF)** if whenever an FD $X \rightarrow A$ holds in R , then X is a **superkey** of R .

- Difference between 3NF and BCNF is that
 - for a functional dependency $A \rightarrow B$, 3NF allows this dependency in a relation if B is a primary-key attribute and A is not a candidate key.
 - Whereas, BCNF insists that for this dependency to remain in a relation, A must be a candidate key.
- Every relation in BCNF is also in 3NF. However, a relation in 3NF is not necessarily in BCNF.
 - That is, BCNF is a stricter form.

A Relation that is in 3NF but not in BCNF

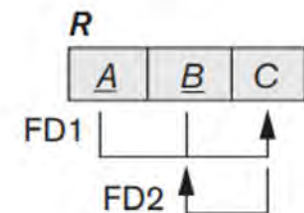
FD1: $\{Student, Course\} \rightarrow Instructor$

FD2: $Instructor \rightarrow Course$

TEACH

<u>Student</u>	<u>Course</u>	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

Dependency $Instructor \rightarrow Course$ holds,
but the determining attribute (*Instructor*)
is neither a *candidate key* nor a *superset of a candidate key*.



Achieving the BCNF by Decomposition

The relation *not* in BCNF should be decomposed:

TEACH_COURSE

<u>Instructor</u>	Course
Mark	Database
Navathe	Database
Ammar	Operating Systems
Schulman	Theory
Mark	Database
Ahamad	Operating Systems
Omiecinski	Database
Navathe	Database
Ammar	Operating Systems

TEACH_STUDENT

<u>Instructor</u>	<u>Student</u>
Mark	Narayan
Navathe	Smith
Ammar	Smith
Schulman	Smith
Mark	Wallace
Ahamad	Wallace
Omiecinski	Wong
Navathe	Zelaya
Ammar	Narayan

Both relations are in BCNF.

Other Normal Forms

- **4NF**: A relation schema R is in 4NF with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every nontrivial *multivalued dependency* $X \twoheadrightarrow Y$ in F X is a *superkey* for R .
- **5NF**: Every non-trivial *join dependency* in the table is implied by the *superkeys* of the table.
- **Domain-key NF**: Every constraint on the table is a *logical consequence* of the table's domain constraints and key constraints.

For more details see: Elmasri, Navathe: Fundamentals of Database Systems

Normalization

EXAMPLE

Example – Simplified CarInsurance DB

- Data to be stored:
 - Registration number
 - Chassis number
 - Model of vehicle (e.g. Ford Focus, Mazda 5)
 - Year of manufacture
 - Colour of vehicle
 - Cubic capacity of the car engine
 - Owner's name
 - Identity card number
 - Address (home)
 - Phone numbers (only one phone number per owners)
 - Name of insurer
 - Address of insurer
 - Phone number of insurer (only one phone number per insurers)
 - Type of insurance (e.g. car liability insurance, comprehensive insurance)
 - Annual fee
 - Sales manager

Simplified CarInsurance DB – 1NF

Put all attributes into a table:

CAR_INSURANCE(RegNum, ChassisNum, Model, MaufactYear, Colour, CubicCapacity, Owner, IdentityCardNum, OwnerAddress, OwnerPhone, Insurer, InsurerAddress, InsurerPhone, InsuranceType, AnnualFee, SalesManager)

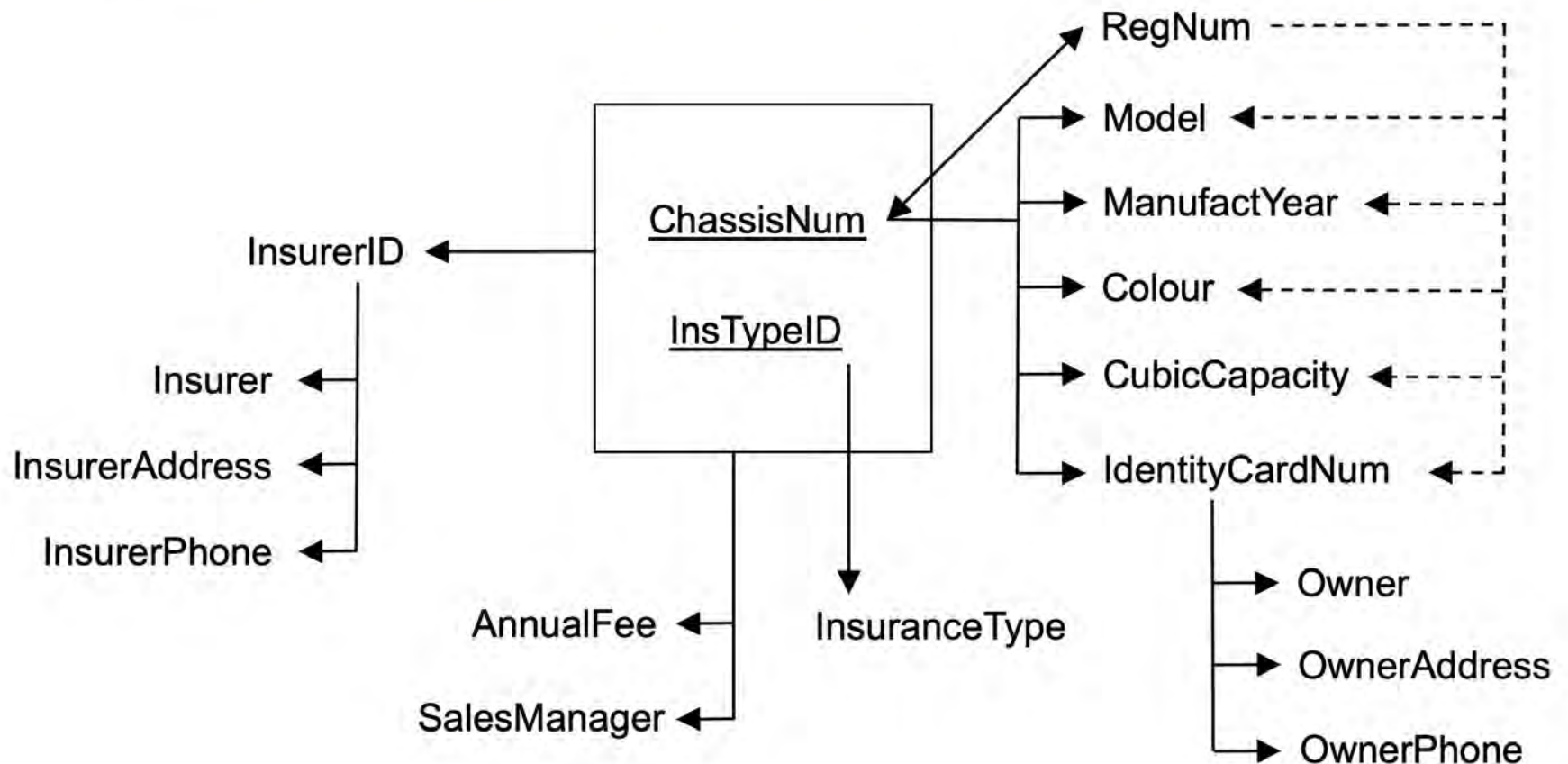
Encode the entities (InsurerID, InsTypeID), and find the primary key:

1NF:

CAR_INSURANCE(RegNum, ChassisNum, Model, MaufactYear, Colour, CubicCapacity, Owner, IdentityCardNum, OwnerAddress, OwnerPhone, *InsurerID*, Insurer, InsurerAddress, InsurerPhone, InsTypeID, InsuranceType, AnnualFee, SalesManager)

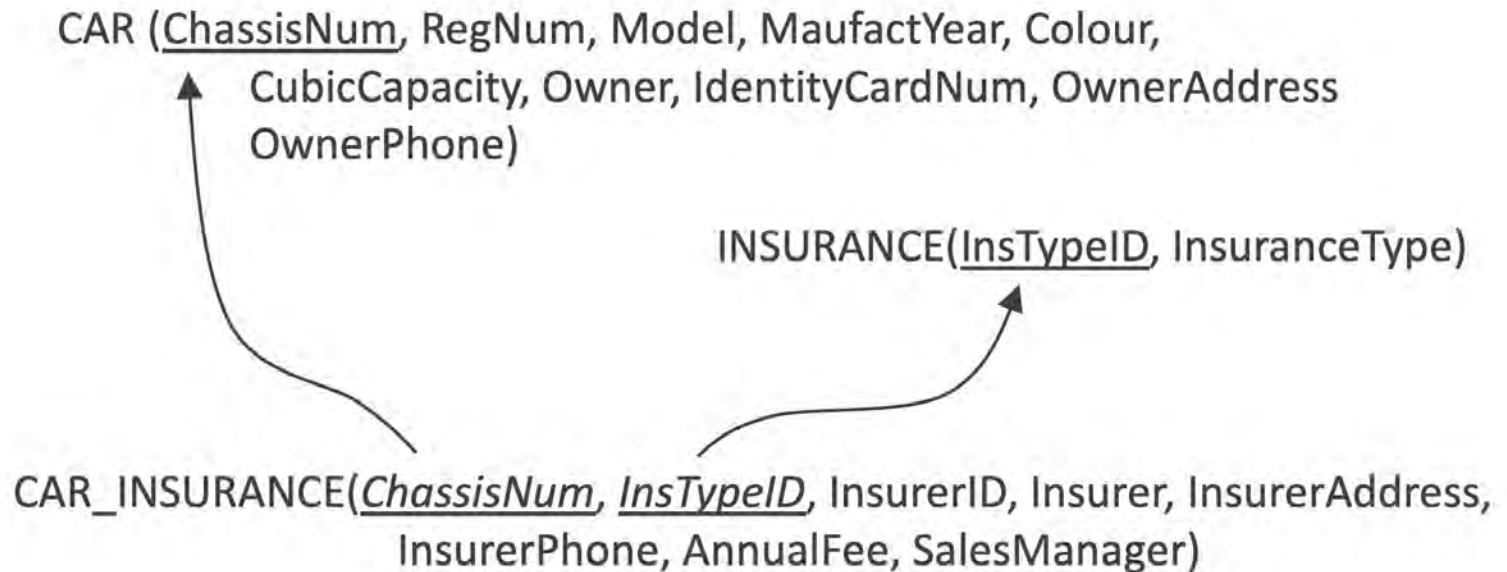
Simplified CarInsurance DB

Draw the **functional dependency (FD) diagram**.



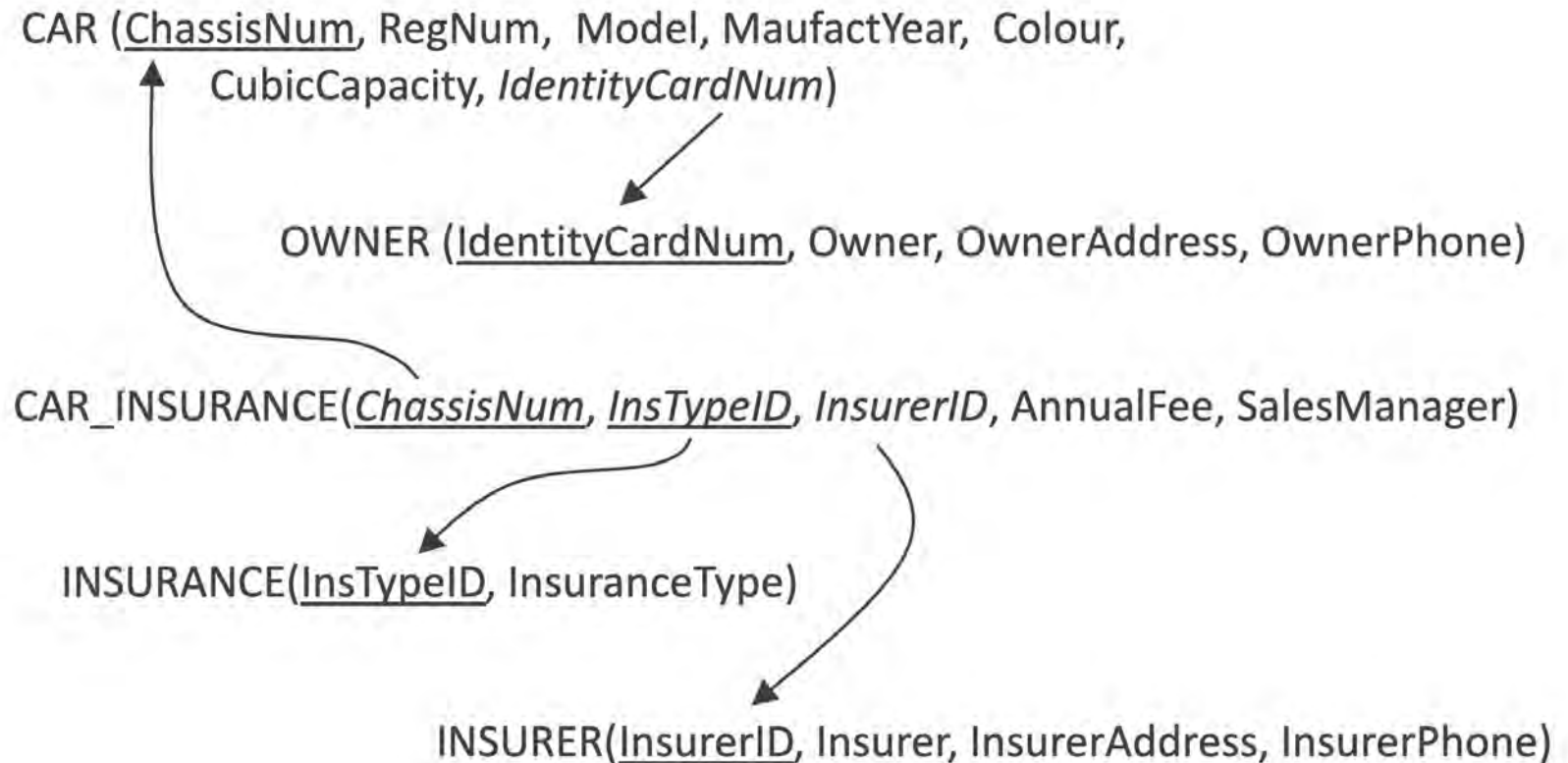
Simplified CarInsurance DB – 2NF

2NF:



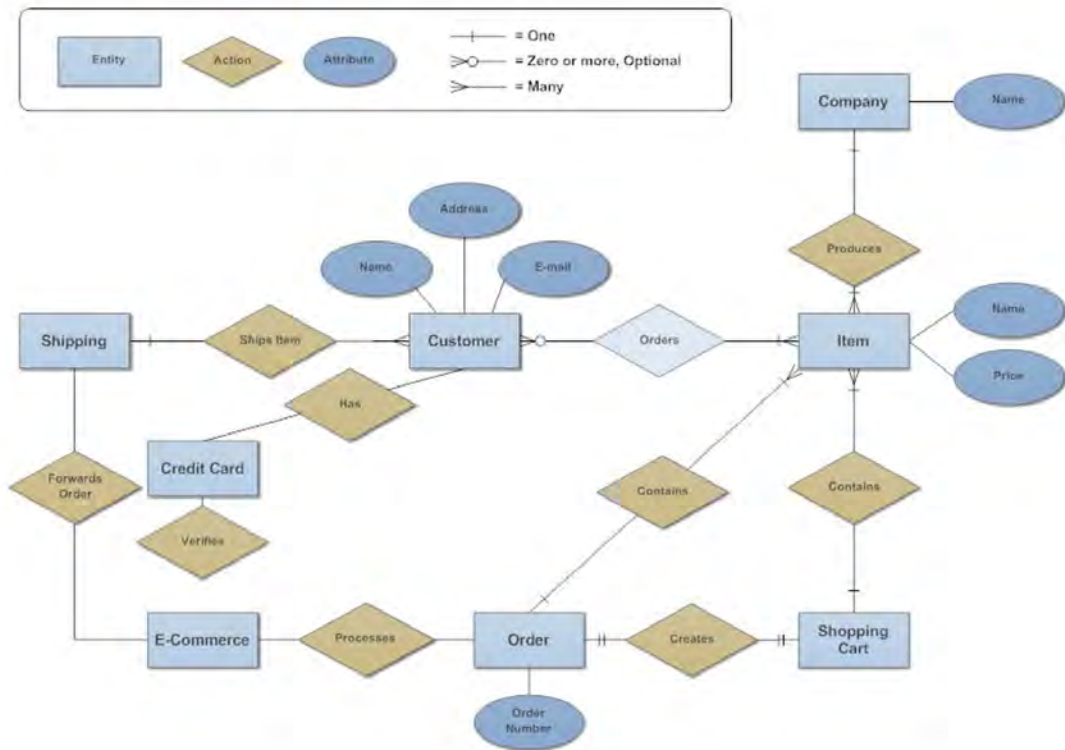
Simplified CarInsurance DB – 3NF

3NF:



Simplified CarInsurance DB - Keys

- Relation CAR:
 - PK: *ChassisNum*
 - FK: *IdentityCardNum* references to relation OWNER
- Relation OWNER:
 - PK: *IdentityCardNum*
- Relation CAR_INSURANCE:
 - PK (composite): *ChassisNum + InsTypeID*
 - FK1: *ChassisNum* references to relation CAR
 - FK2: *InsTypeID* references to relation INSURANCE
 - FK3: *InsurerID* references to relation INSURER
- Relation INSURANCE:
 - PK: *InsTypeID*
- Relation INSURER:
 - PK: *InsurerID*



Entity-Relationship Model

Table of Content

- ER model concepts
 - Entities, entity types and attributes
 - Relationships and relationship types
 - Weak entity types
- ER diagram
- Example

Concepts of the ER Model

- A database can be modelled as:
 - a collection of *entities*, and
 - *relationships* among entities
- Three main concepts of the Entity-Relationship Model:
 - entity types
 - attributes
 - relationship types
- **Entity-Relationship (ER) Diagram**
 - A *graphical representation* of the entities and associations between them.

Concepts of the ER Model

- An **entity** is an object that is distinguishable from other objects.
 - Example: a specific person (e.g. John Smith), a specific university (e.g. University of Pannonia), a specific event (e.g. database exam)
- Entities have **attributes**.
 - attributes are properties to describe an entity
 - Example: people have *names* and *addresses*
- An **entity type (entity set)** is a set of entities of the same type that share the same properties.
 - Example: set of all persons, set of universities, set of events

Examples – Entity type, Attributes, Entity

- **entity type:** PERSON
 - a set of people
 - it has attributes for example:
 - *Name*,
 - *SSN*,
 - *Sex*,
 - *BirthDate*
- **entity:** a specific person, e.g.:
 - *Name*=*John Smith*,
 - *SSN*=*'AB123456789'*,
 - *Sex*=*'M'*,
 - *BirthDate*= *'12-FEB-1970'*

Types of Attributes

- **Simple attribute**

- the attribute has a *single atomic value* (or NULL value)
 - e.g. *SSN* or *Sex*

- **Composite attribute**

- the attribute may be *composed of several components*
 - e.g. *Address* (*House#*, *Street*, *City*, *ZipCode*, *Country*),
or *Name* (*FirstName*, *LastName*)

- **Multivalued attribute**

- an entity *may have multiple values* for that attribute
 - e.g. *Phone#* of a *PERSON*

Types of Attributes

- **Derived attribute**

- the value of the attribute is *derivable* from value(s) of related attribute(s)
 - related attributes are not necessarily in the same entity type
 - e.g. *Age* is derivable from *BirthDate*

- In general, composite and multi-valued attributes may be nested arbitrarily to any number of levels.
 - e.g. *PreviousDegrees* of a STUDENT is a *composite multi-valued attribute* denoted by {*PreviousDegrees (College, Year, Degree, Field)*}.

Types of Attributes

- **Candidate Key**

- Minimal set of attributes that *uniquely identifies* each occurrence of an entity type.
 - e.g. *SSN*, or *IdentityCardNum*, for PERSON

- **Key Attribute**

- Candidate *key selected to uniquely identify* each occurrence of an entity type.

- **Composite Key Attribute**

- A primary key that consists of two or more attributes.

ER Diagram for Entity Types

- An entity type is shown as a rectangle labeled with the name of the entity.
 - The name is usually a singular noun.



ER Diagram for Attributes

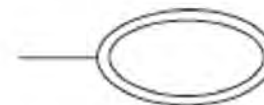
- Attributes are displayed in ovals.
 - each attribute is connected to its entity type



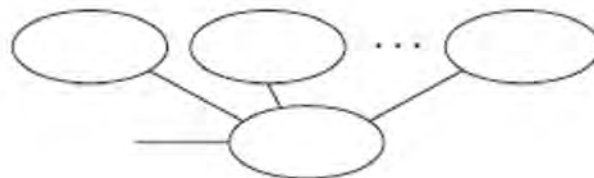
Attribute



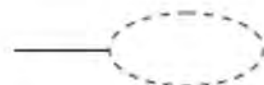
Key Attribute



Multivalued Attribute

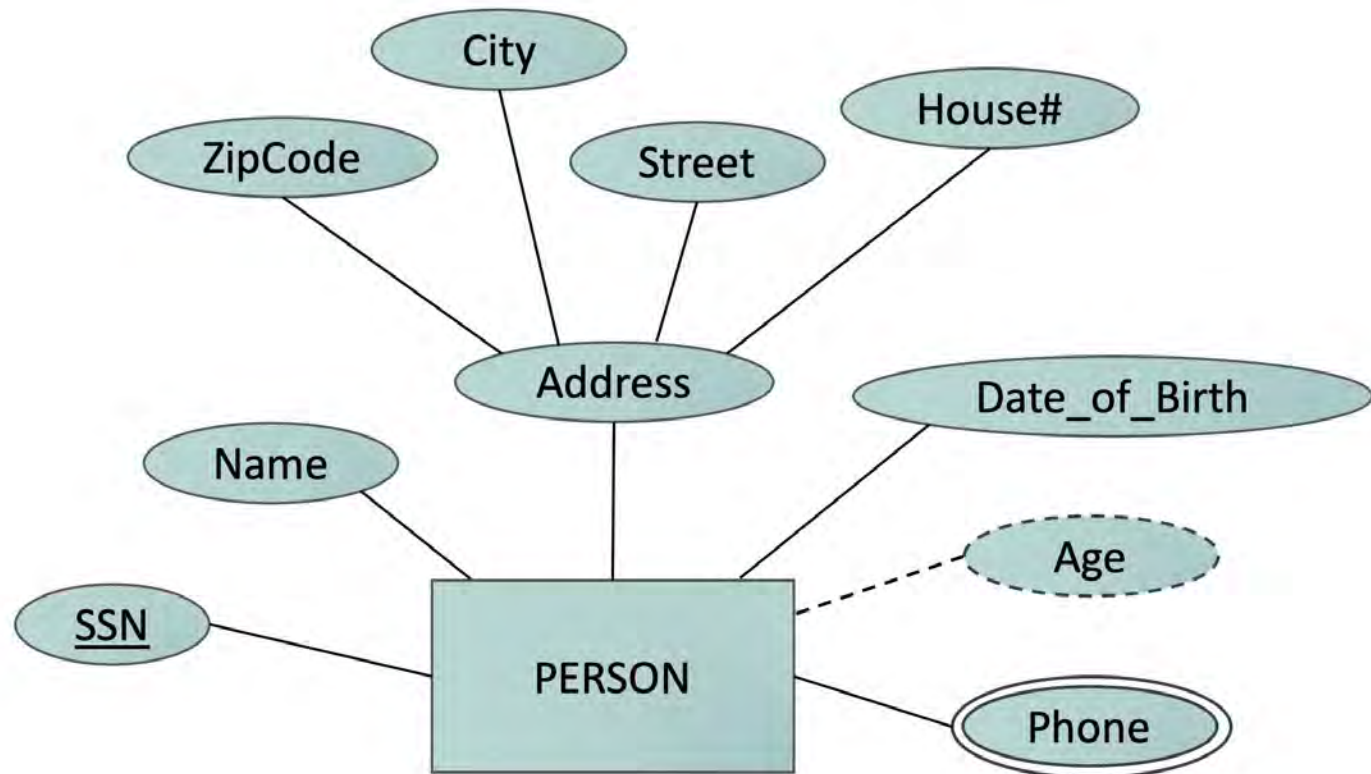


Composite Attribute

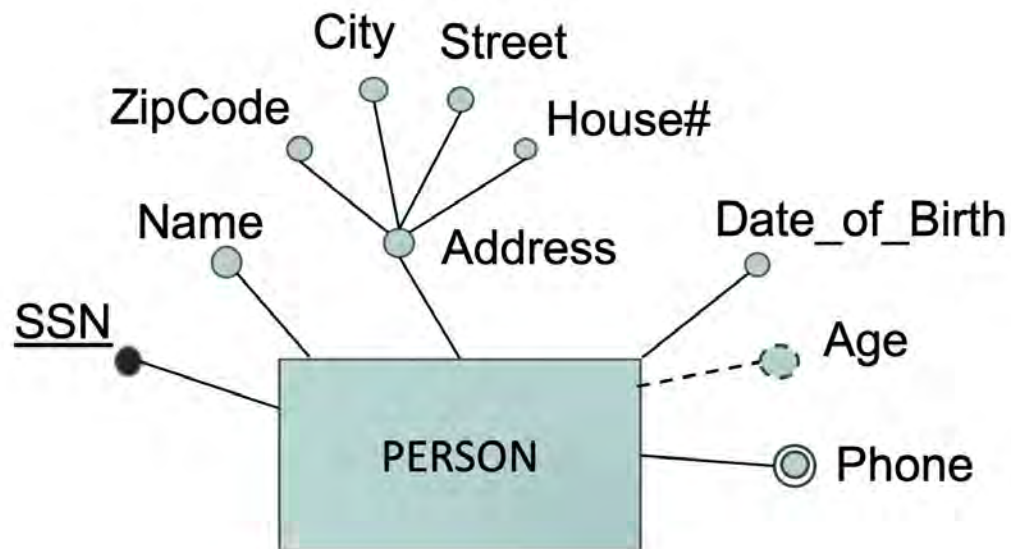


Derived Attribute

Example – Entity Type with Attributes



Example – Alternative Notation



Relationship Types

- **Relationship type**

- Set of meaningful associations among entity types.
- Each relationship type is given by a *name* that describes its function.
- e.g. PERSON *has* a CAR, EMPLOYEE *works on* a DEPARTMENT

- **Relationship occurrence (relationship instance)**

- Uniquely identifiable association, which includes one occurrence from each participating entity type.
-
- We can examine examples of individual occurrences of a relationship using **semantic nets**.

Degree of Relationships

- The entities involved in a particular relationship type are referred to as **participants**.
- The **degree of a relationship** is the *number of participating entities* in a relationship.
- Based on the number of participants relationships are categorized as:
 - 1: **recursive**
 - 2: **binary**
 - 3: **ternary**
 - *n*: ***n*-ary**

Cardinality Constraints on Relationships

- **Maximum Cardinality**

- **One-to-one (1:1)**

- In a one-to-one relationship, each entity of either entity set is related to at most one entity of the other set.
 - e.g. HUSBAND – *has* – WIFE relationship (according to the Christian religion at one point in time)

- **One-to-many (1:N) or Many-to-one (N:1)**

- Each entity of the first set is connected to at most one entity of the second set.
 - But an entity of the second set can be connected to zero, one, or many entities of the first set.
 - e.g. OWNER – *has* – CAR relationship

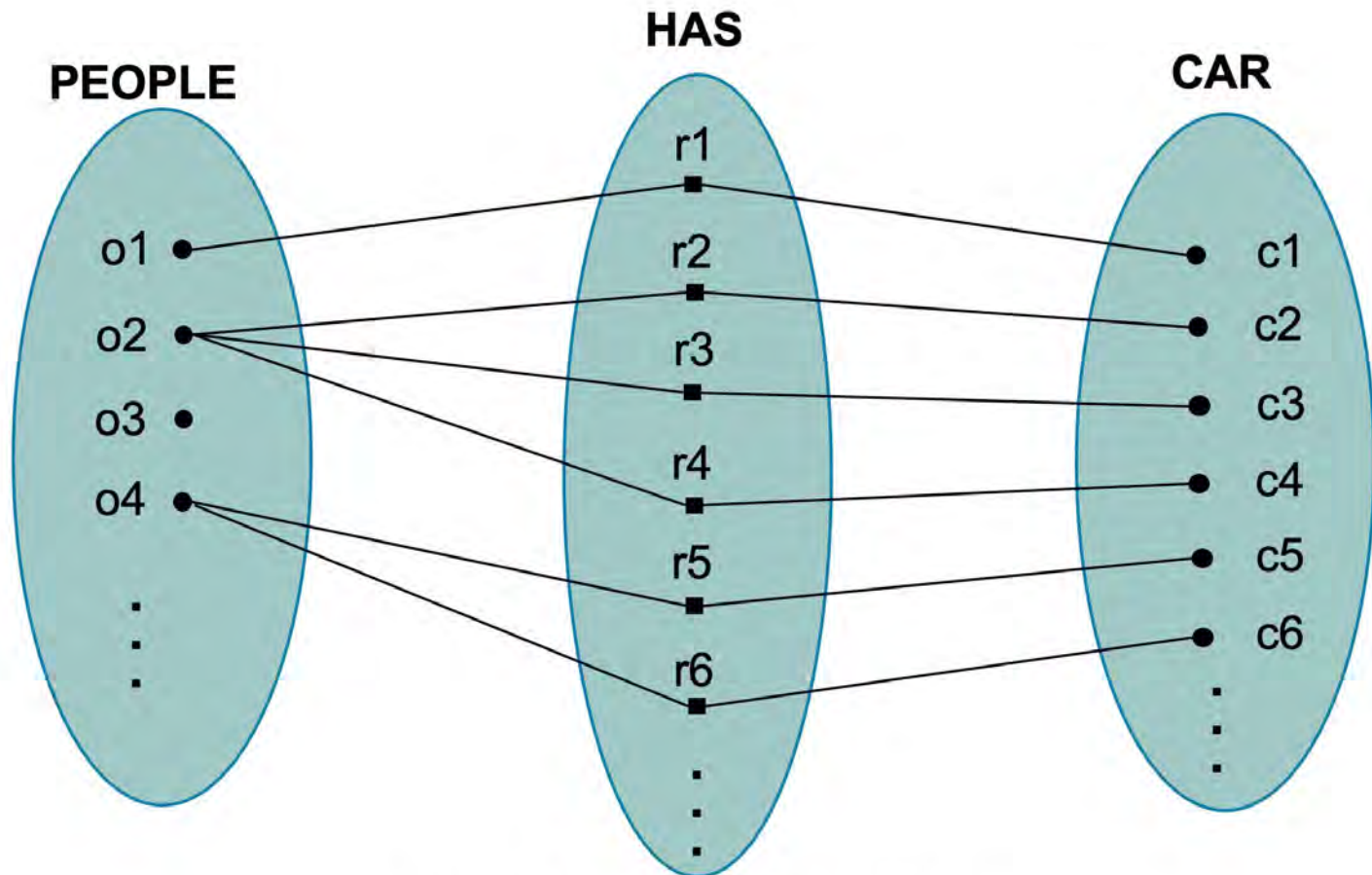
- **Many-to-many (M:N)**

- In a many-to-many relationship, an entity of either set can be connected to many entities of the other set.
 - e.g. STUDENT – *enrolls* – COURSE relationship

Cardinality Constraints on Relationships

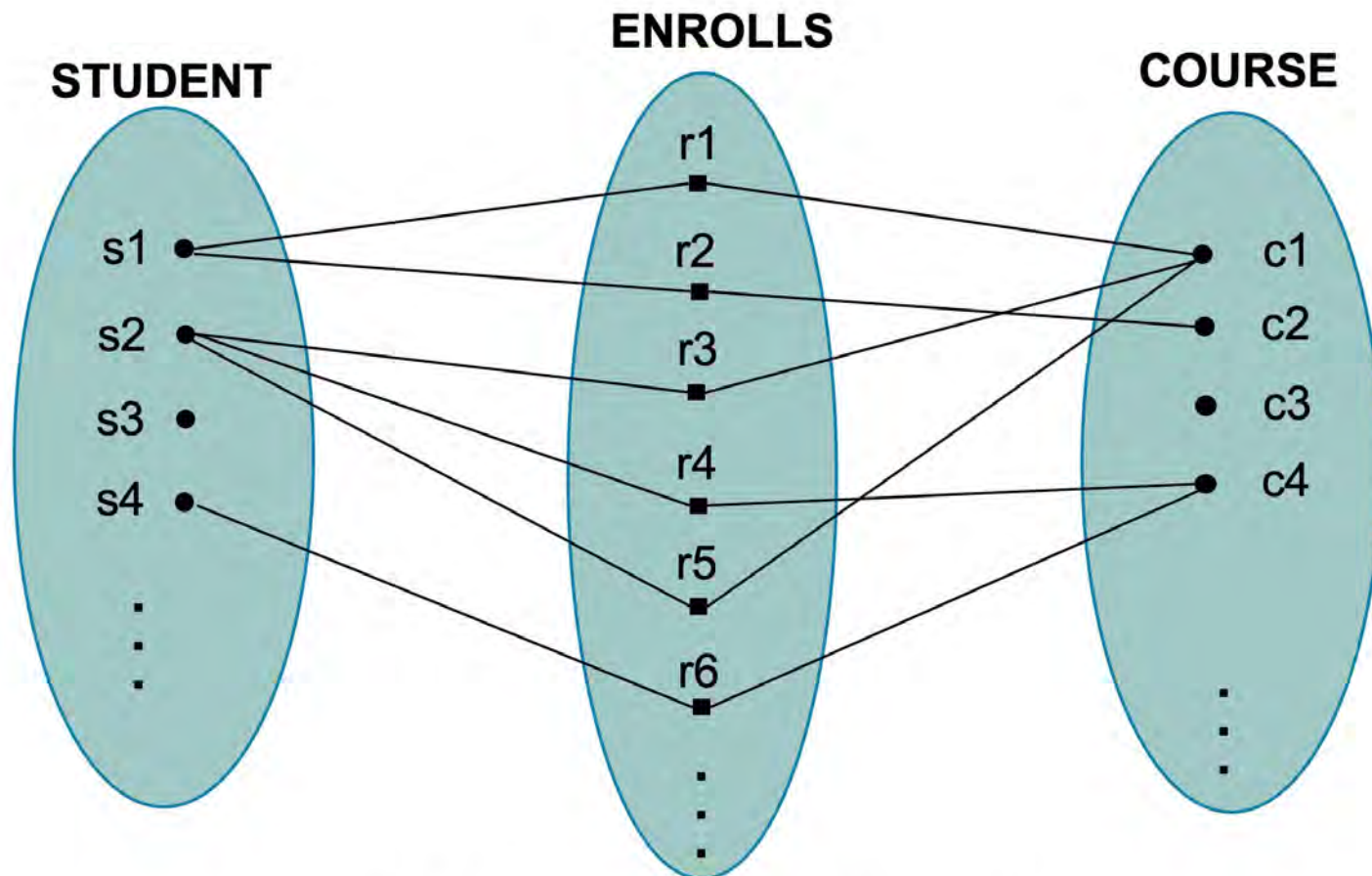
- **Minimum Cardinality:** The minimum cardinality of a relationship is the minimum number of instances of an entity B that may be associated with each instance of an entity A .
 - zero: **partial** or **optional participation**
 - one or more: **total participation**

Example – 1:N Cardinality (Semantic Net)



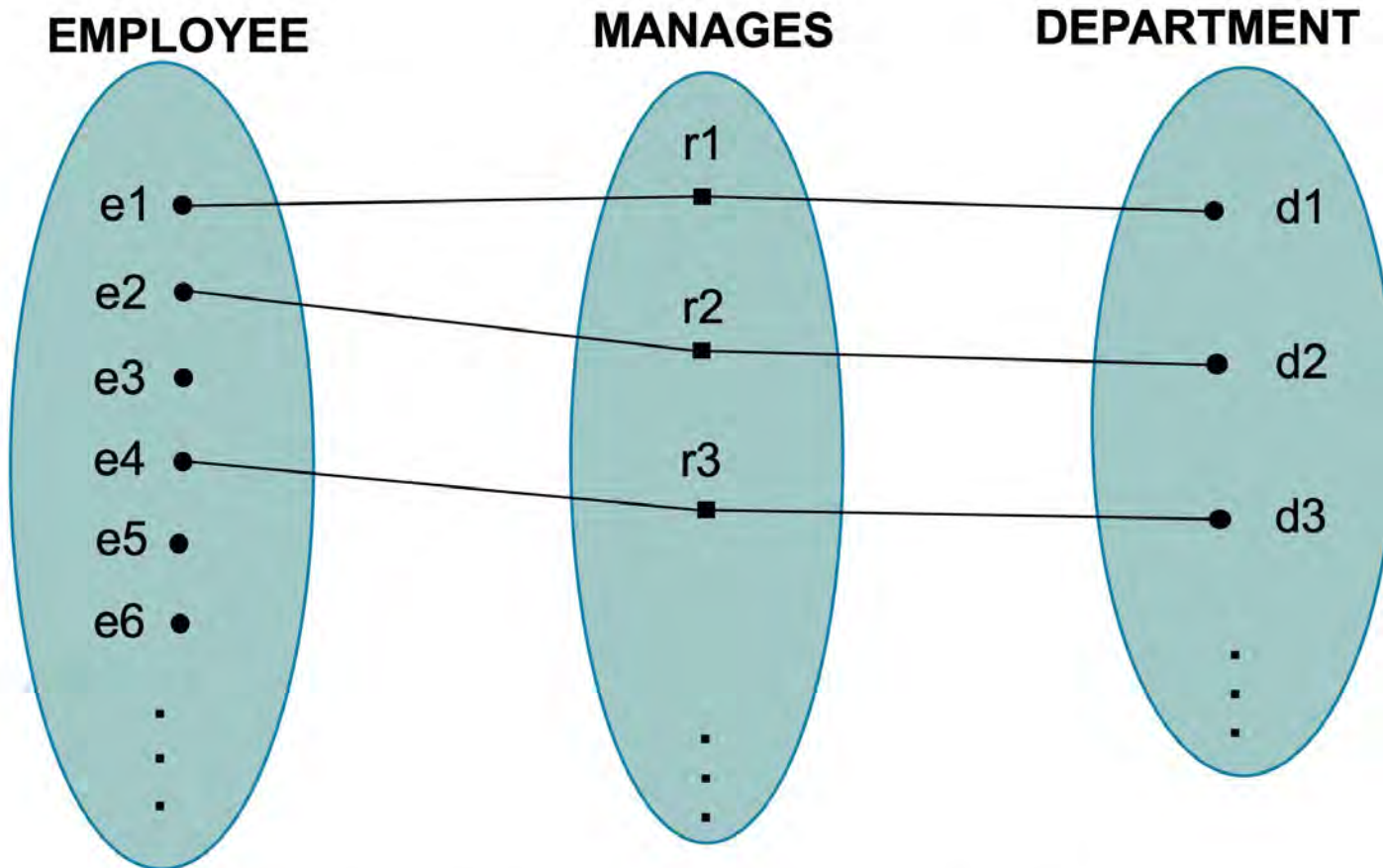
HAS 1:N relationship between
PEOPLE (partial participation) and CAR (total participation) entities

Example – M:N Cardinality (Semantic Net)



ENROLLS M:N relationship between
STUDENT (partial participation) and COURSE (partial participation) entities

Example – 1:1 Cardinality (Semantic Net)

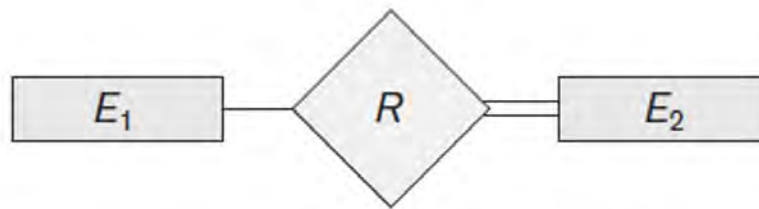


MANAGES 1:1 relationship between
EMPLOYEE (partial participation) and DEPARTMENT (total participation) entities

ER Diagram for Relationship Type

- In ER diagrams, we represent the *relationship type* as follows:
 - *diamond-shaped box* is used to display a relationship type,
 - *connected to* the participating *entity types* via straight lines,
 - *minimum* and *maximum constraints* of the relation type are denoted on the ER diagram.

ER Diagram for Relationship Type

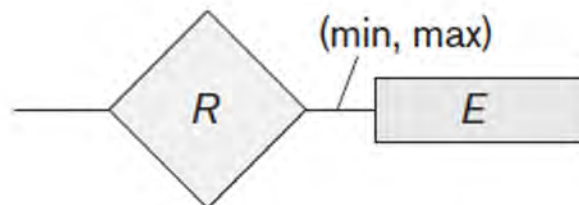


Total Participation of E_2 in R



Cardinality Ratio 1: N for $E_1:E_2$ in R

Alternative notation (min-max notation):



Structural Constraint (min, max)
on Participation of E in R

Example



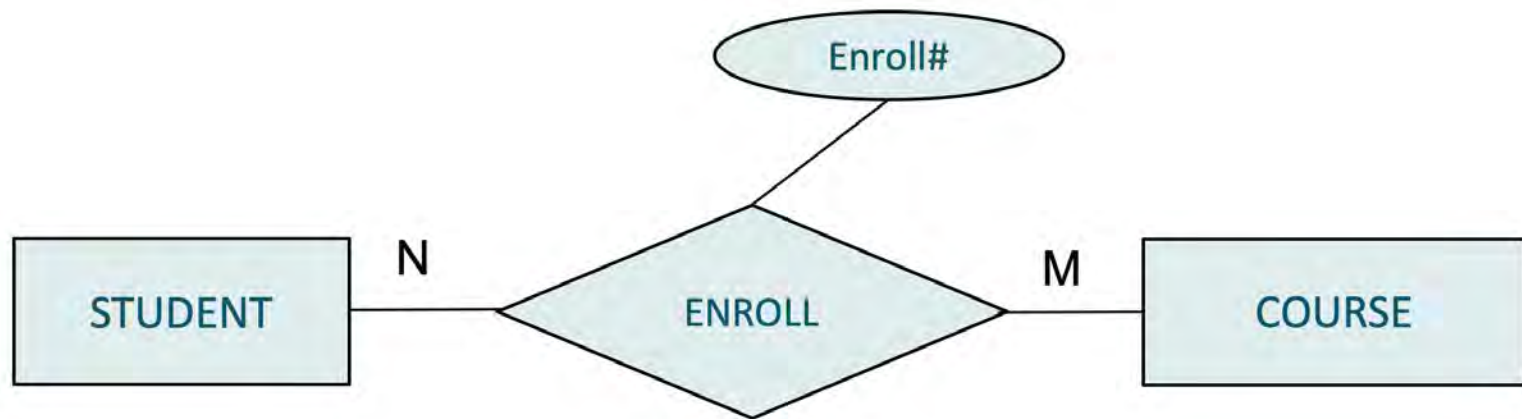
Alternative notation:



Attributes of Relationship Types

- Relationship types can also have attributes.
 - For example, include the *Start_date* attribute for the MANAGES relation type. This attribute yields the date on which a manager started to manage a department.
 - Most relationship attributes are used with M:N relationships. In 1:N relationships, they can be transferred to the entity type on the N-side of the relationship.

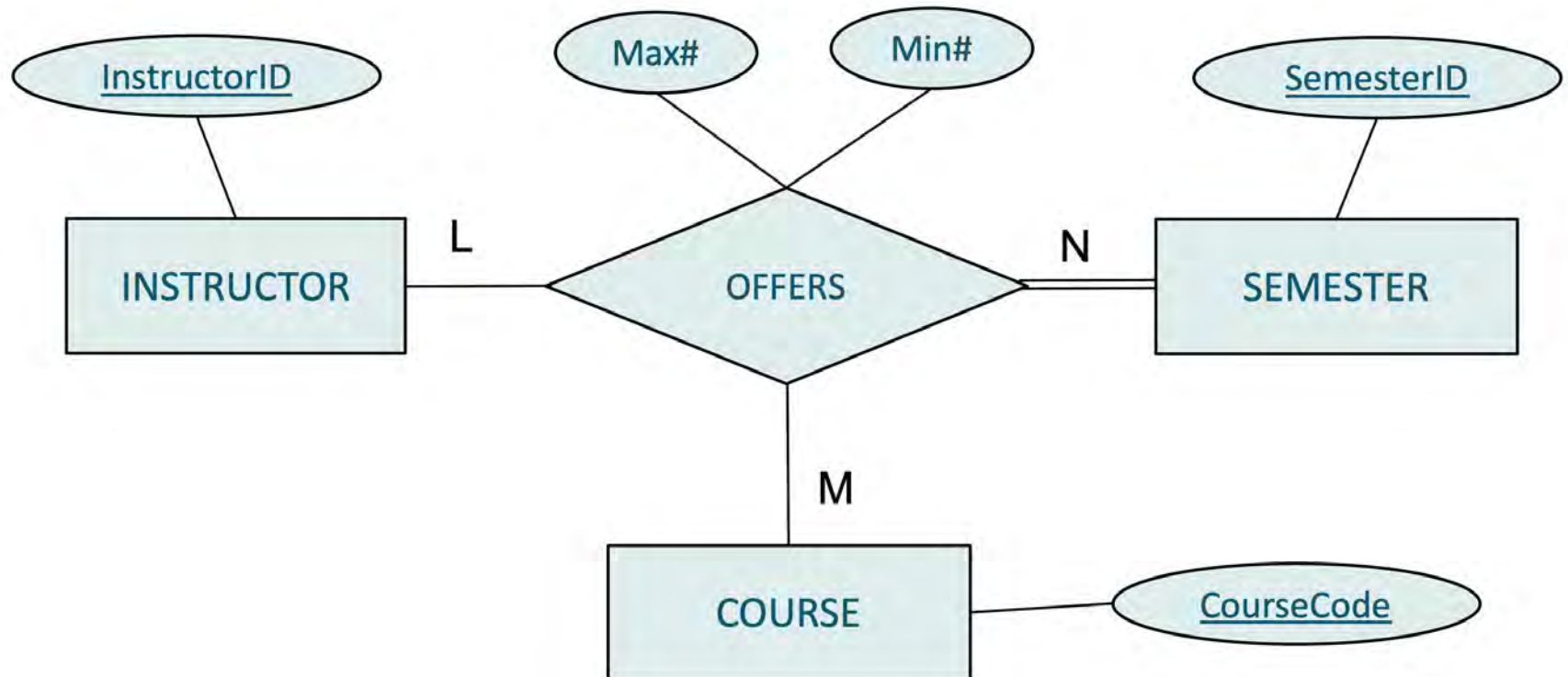
Example – Attribute of a Relationship Type



Relationships with Higher Degree

- In general, an *n-ary* relationship is not equivalent to n binary relationships!
- Constraints are harder to specify for higher-degree relationships ($n > 2$) than for binary relationships.

Example – Relationship with Higher Degree

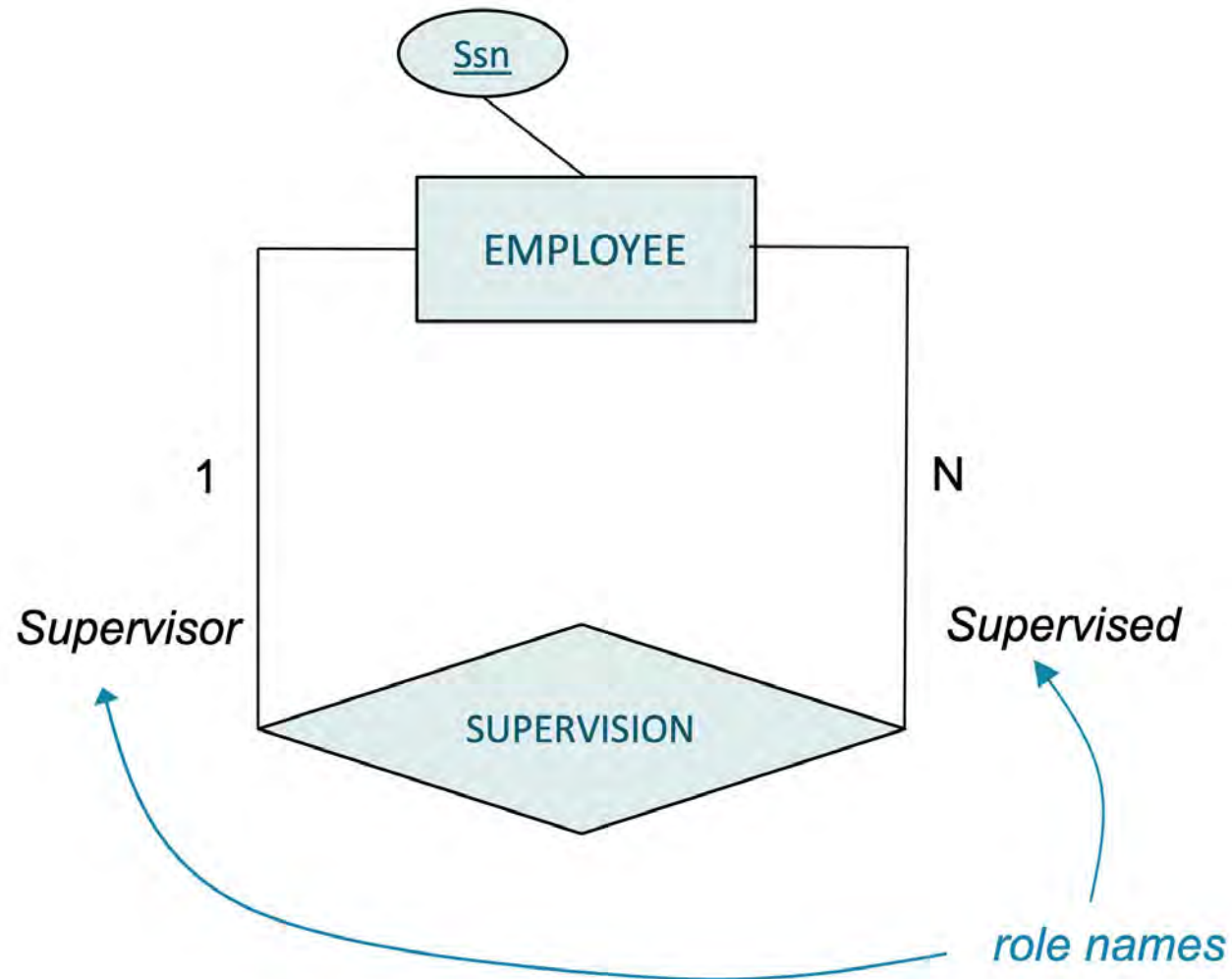


Recursive Relationship

recursive relationship type:

- Both participations are same entity type in different roles.
- For example, SUPERVISION relationships between EMPLOYEE (in role of supervisor or boss) and (another) EMPLOYEE (in role of subordinate or worker).
- In ER diagram, *need to display role names* to distinguish participations.

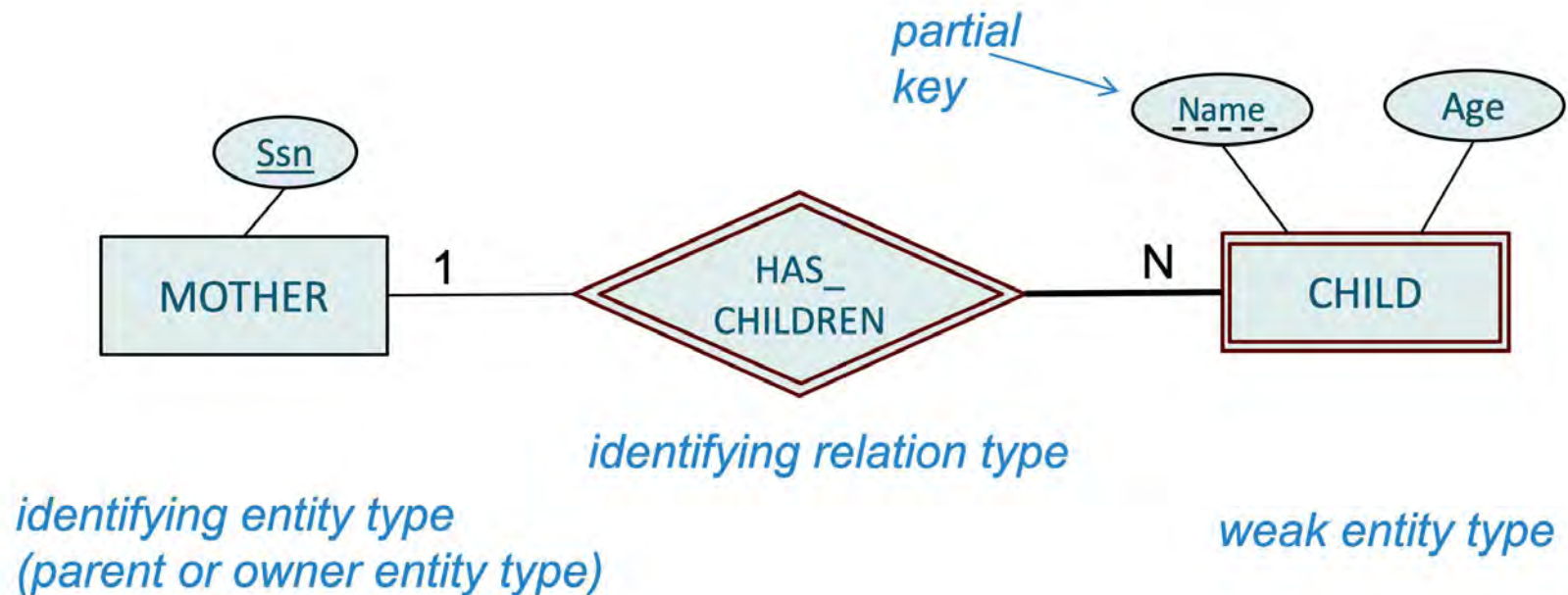
Example – Recursive Relationship



Weak Entity Types

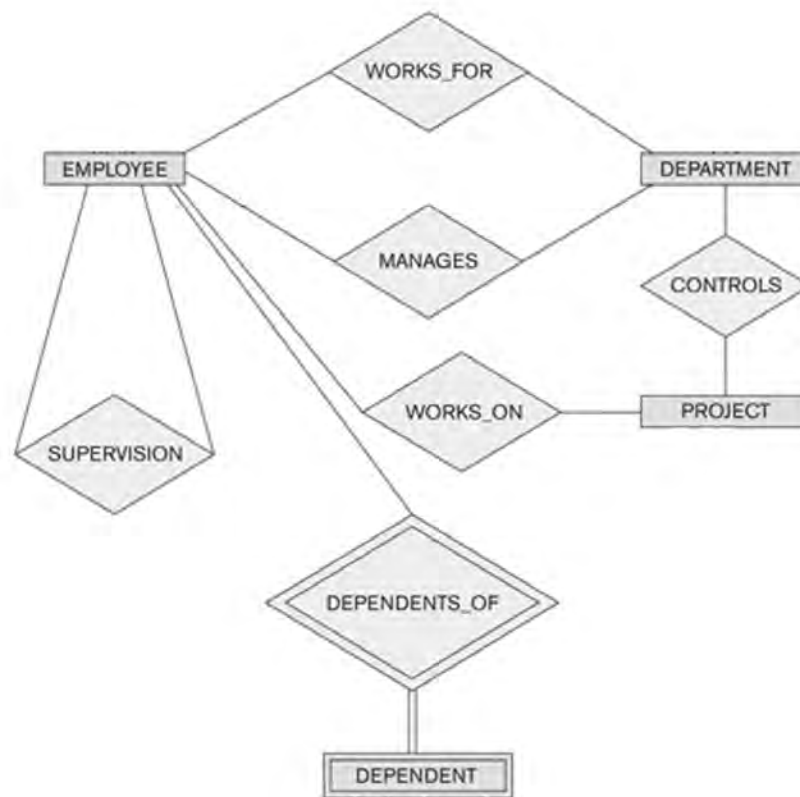
- **Weak entity type**: an entity type that does not have a key attribute
 - *must participate* in an **identifying relationship**
 - entities of a weak entity type are identified by the combination of:
 - a **partial key** of the weak entity type, and
 - **Partial key**: An attribute that when combined with the primary key of another entity type makes this entity unique.
 - the **particular entity** that is related to in the identifying entity type
 - Notations in ER Model:
 - weak entity: rectangle using a *double line*
 - partial key: attribute name is *underlined with dashed line*
 - identifying relationship type: *double line diamond*

Example – Weak Entity Type

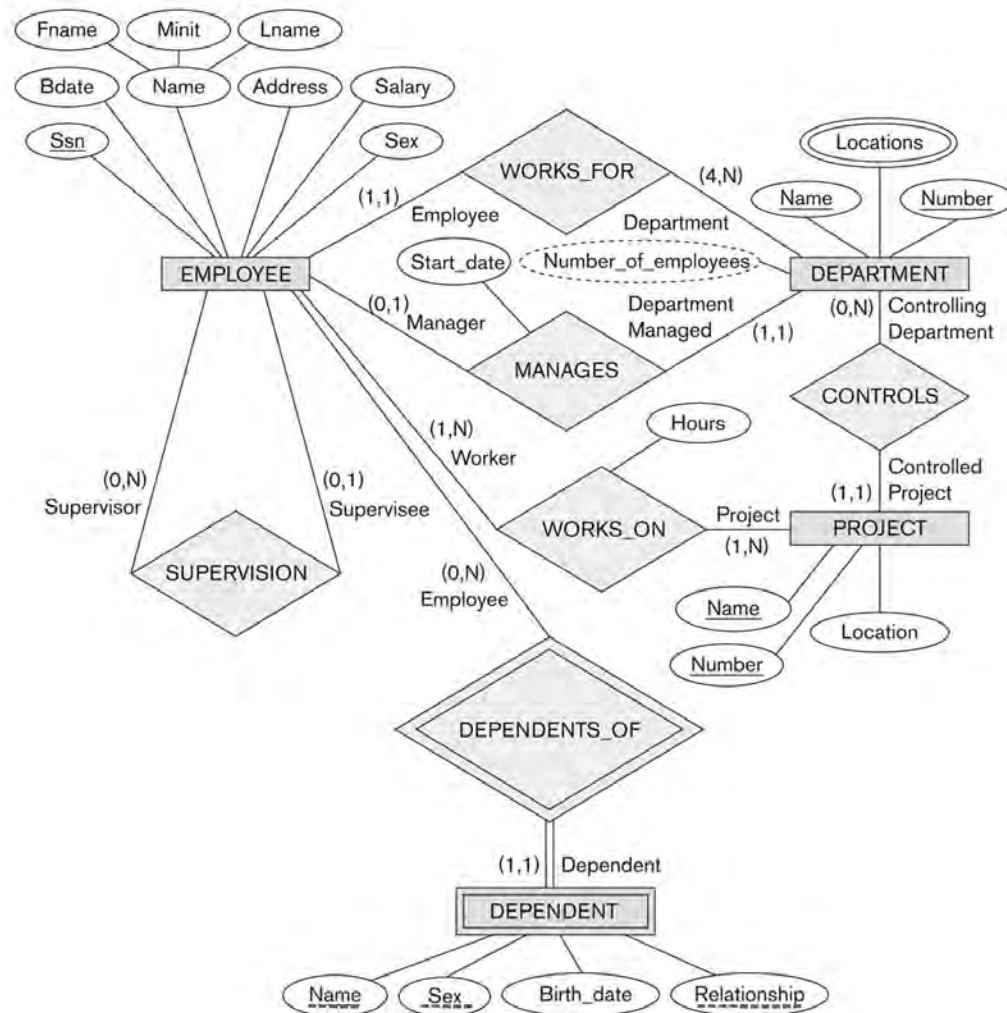


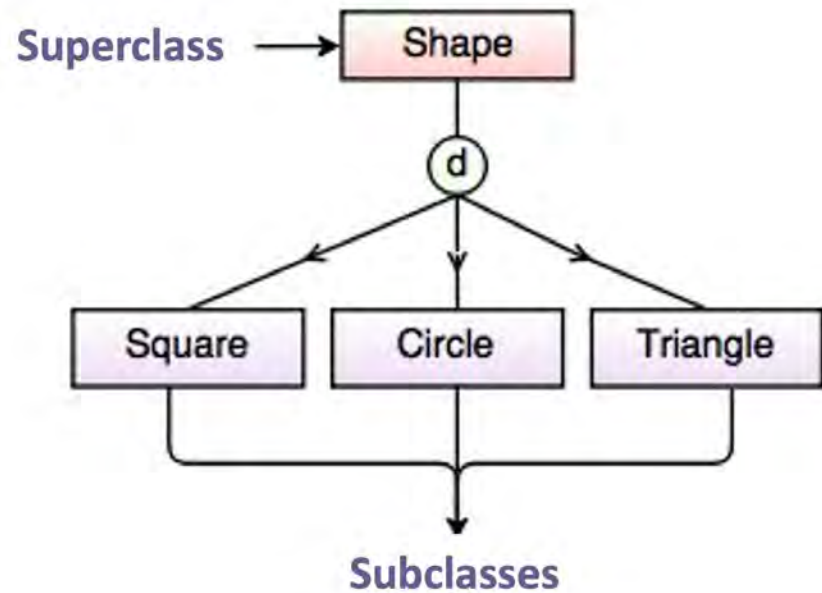
Example – Company Database (Skeleton Schema)

Skeleton schema: (E)ER model without attributes



Example – Company Database (ER Model)





Enhanced Entity-Relationship Model

Table of Content

- EER model concepts
- Alternative diagrammatic notations
- Problems with ER model

Enhanced Entity-Relationship (EER) Model

- EER stands for **Enhanced ER** or **Extended ER**
- **EER Model**
 - includes all modelling concepts of basic ER
 - EER includes some object-oriented concepts, such as inheritance
 - additional concepts:
 - subclasses/superclasses
 - specialization/generalization
 - categories (UNION types)
 - attribute and relationship inheritance

Subgrouping of Entities

- An entity type may have additional meaningful *subgroupings of its entities*.
 - Example: A SHAPE can be *square, circle or triangle*.
- The same entity type can be grouped in different ways based on different aspects:
 - Example: EMPLOYEE may be further grouped into:
 - based on the employee's job: *secretary, engineer, technician, ...*
 - based on leadership: *manager, not manager*
 - based on the payment method: *salaried employee, hourly worker*

Subclass and Superclass

- **Subclass:** A distinct subgrouping of occurrences of an entity type.
- **Superclass:** An entity type that includes one or more distinct subgroupings of its occurrences.
- **Superclass/subclass relationship:** relationship between a superclass and any of its subclasses.
- Examples:
 - The entity type SQUARE is a subclass of the SHAPE superclass.
 - The entity type ENGINEER is a subclass of the EMPLOYEE superclass.
 - Between the entity types ENGINEER and EMPLOYEE, there is a superclass/subclass relationship.

Example – Subclasses and Superclass

- **Superclass:** EMPLOYEE

- it has:

- primary key
 - general attribute(s)

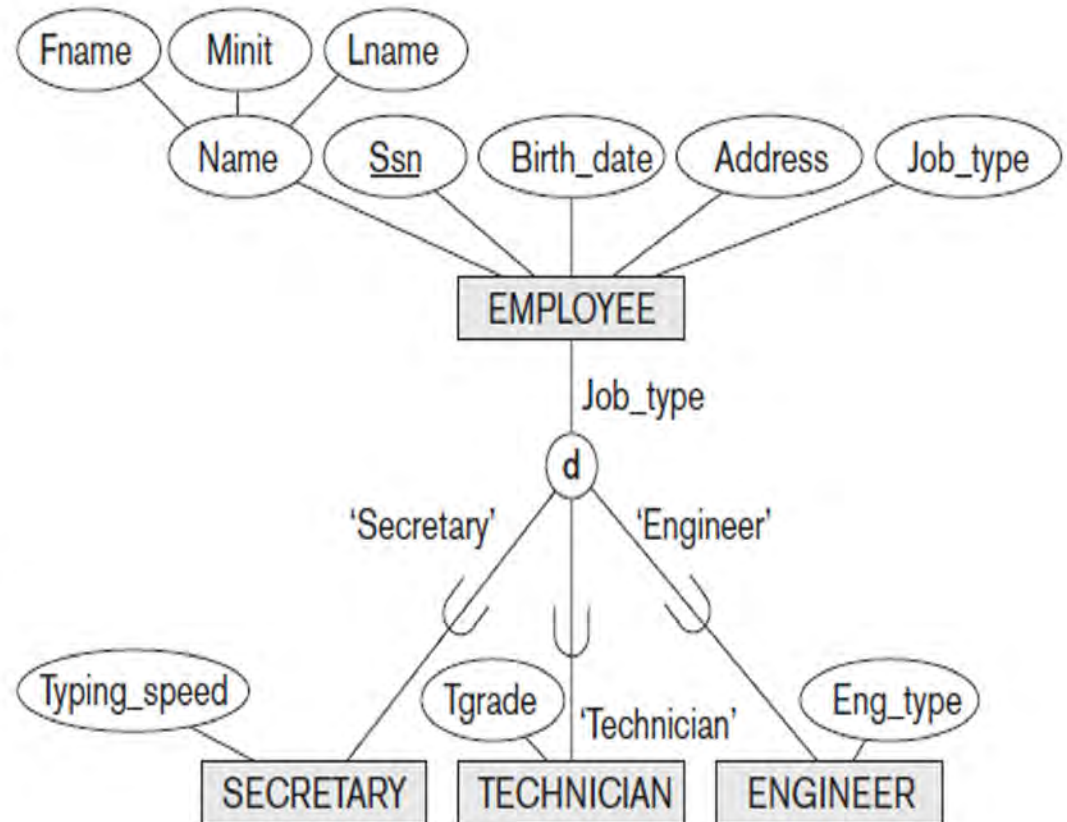
- it may be a part of

- general relationships(s)

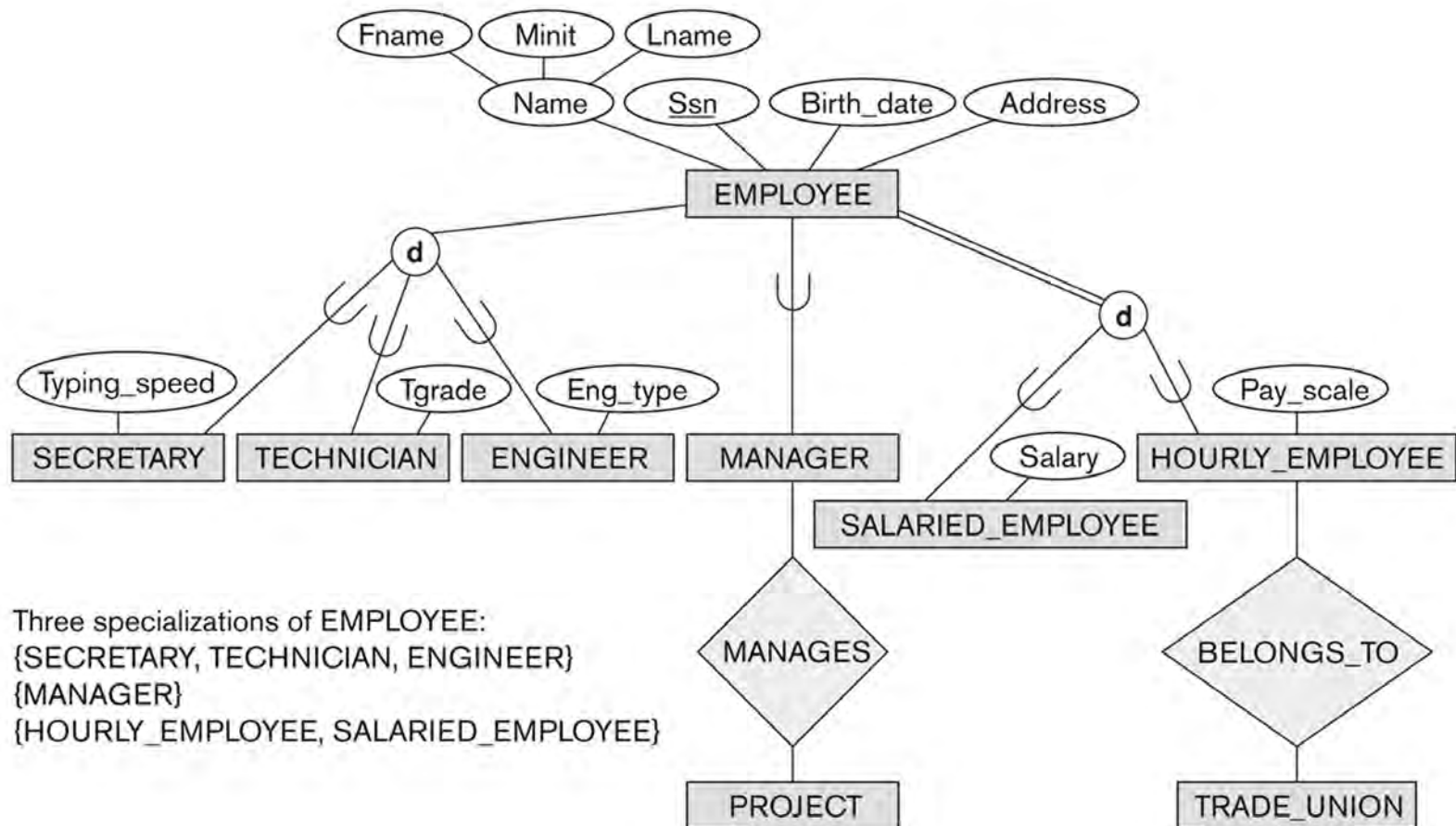
- **Subclasses:** SECRETARY, TECHNICIAN, ENGINEER

- they may have:

- special attribute(s)
 - special relationship(s)



Complex Example – Subclasses and Superclass



Inheritance

An entity that is member of a subclass represents the *same real-world entity* as some member of the superclass:

- An entity can not exist only in the subclass, *it must also be a member of the superclass*.
- It is not necessary that every entity in a superclass be a member of some subclass
- A member of the superclass can be *optionally included* as a member of any number of its subclasses.

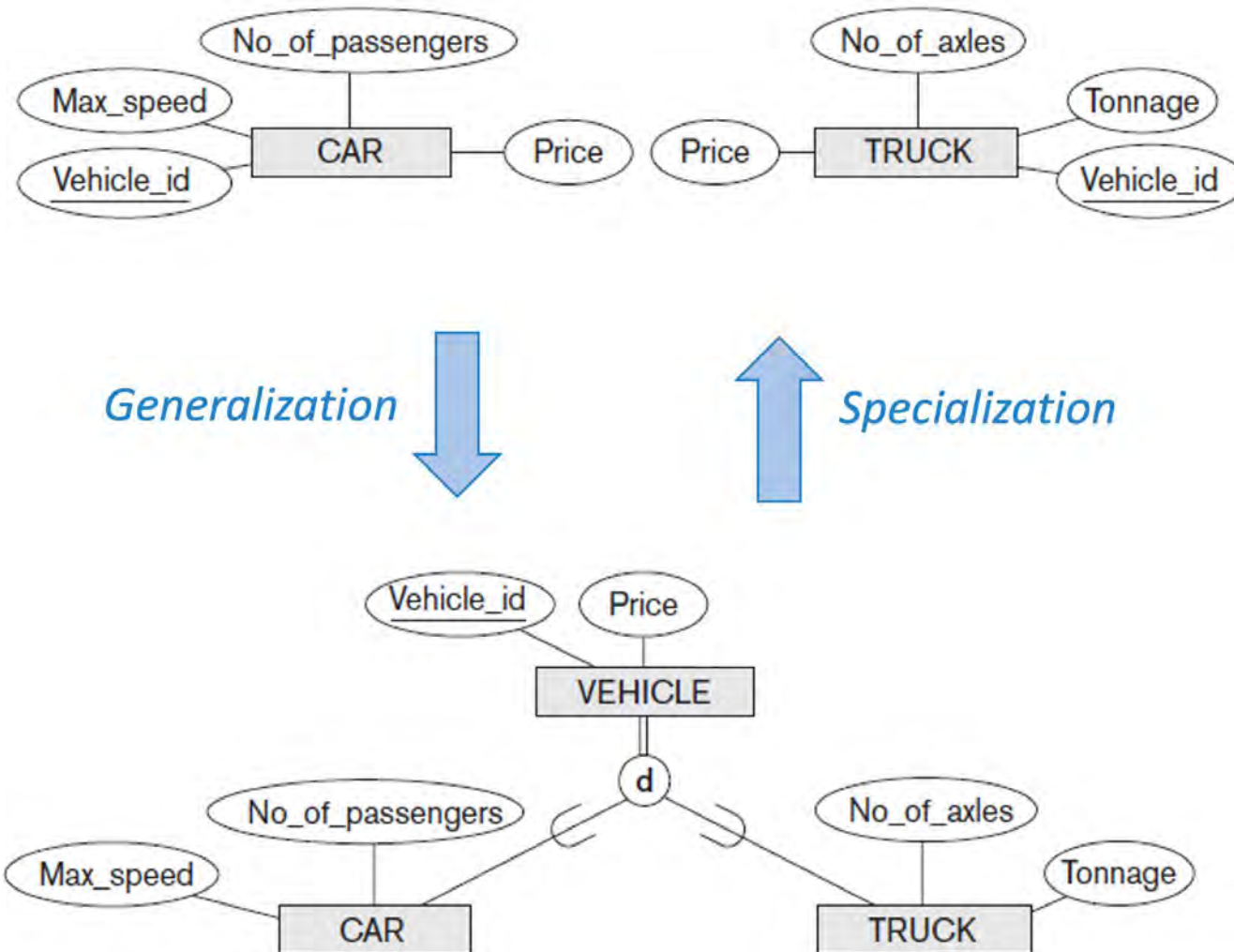
Inheritance:

- An entity that is member of a subclass *inherits*
 - *all attributes* of the entity as a member of the superclass, and
 - *all relationships* of the entity as a member of the superclass.
- Example:
 - SECRETARY inherits the attributes *Name*, *SSN*, ... from EMPLOYEE

Specialization and Generalization

- **specialization:** *top down* conceptual *refinement process*
 - the process of defining a set of subclasses of a superclass based on some distinguishing characteristics of the entities in the superclass
- **generalization:** *bottom up* conceptual *synthesis process*
 - the *reverse of the specialization process*
 - several classes with common features are generalized into a superclass

Example – Generalization



Constraints on Superclass/Subclass Relationships

Types of constraints:

- **Disjointness constraint**

- Determines whether member of a superclass can be a member of one or more than one subclasses.

- **Completeness constraint**

- Determines whether every member in the superclass must participate as a member of a subclass or not.

Constraints on Superclass/Subclass Relationships

- **Disjointness constraint:**

- **disjoint:** specifies that the subclasses **are disjoint**:
 - an entity can be a member of at most one of the subclasses
 - notation: ***d***
- **overlapping:** specifies that the subclasses **may overlap**:
 - the same entity may be a member of more than one subclass
 - notation: ***o***

Constraints on Superclass/Subclass Relationships

- **Completeness Constraint:**

- **Total:** specifies that every entity in the superclass **must be a member** of some subclass
 - notation: **double line**
- **Partial:** allows an entity **not to belong to any** of the subclasses
 - notation: **single line**

Types of Superclass/Subclass Relationships

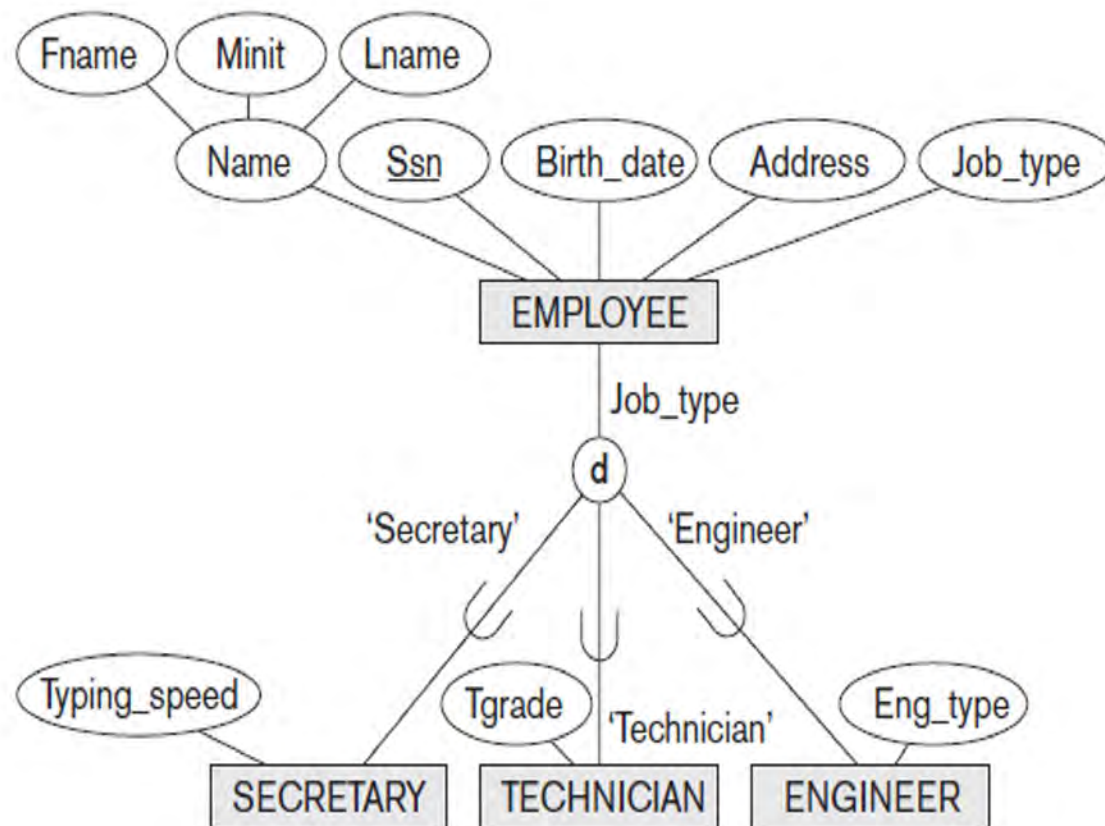
4 types of superclass/subclass relationships:

- disjoint, total
- disjoint, partial
- overlapping, total
- overlapping, partial

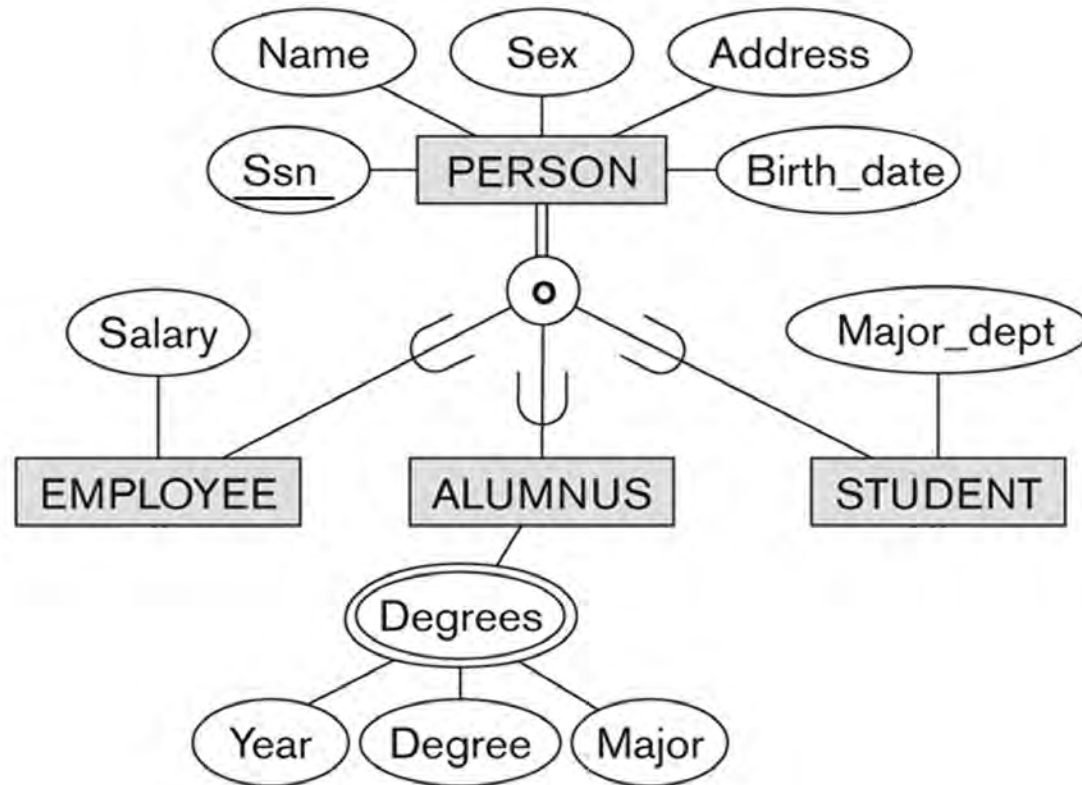
Note:

- Generalization usually is total because the superclass is derived from the subclasses.

Example of Disjoint Partial Superclass/Subclass Relationship



Example of Overlapping Total Superclass/Subclass Relationship

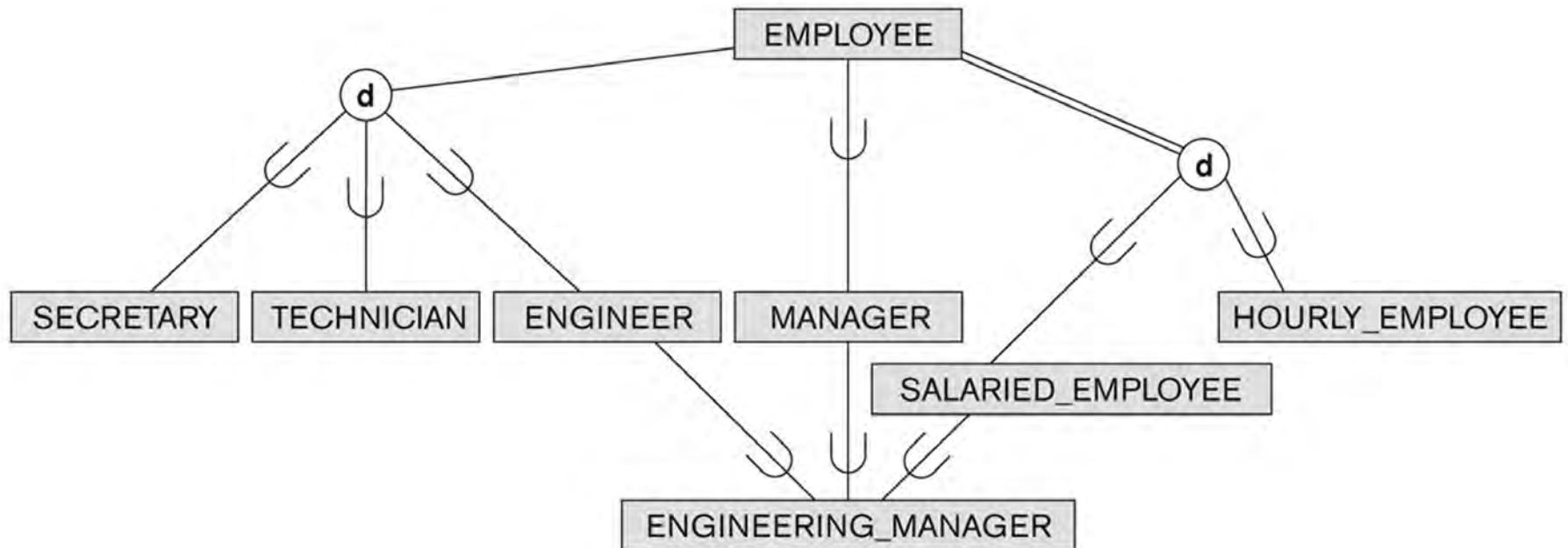


Hierarchies, Lattices and Shared Subclasses

A subclass itself may have further subclasses specified on it, forming a hierarchy or a lattice of specializations.

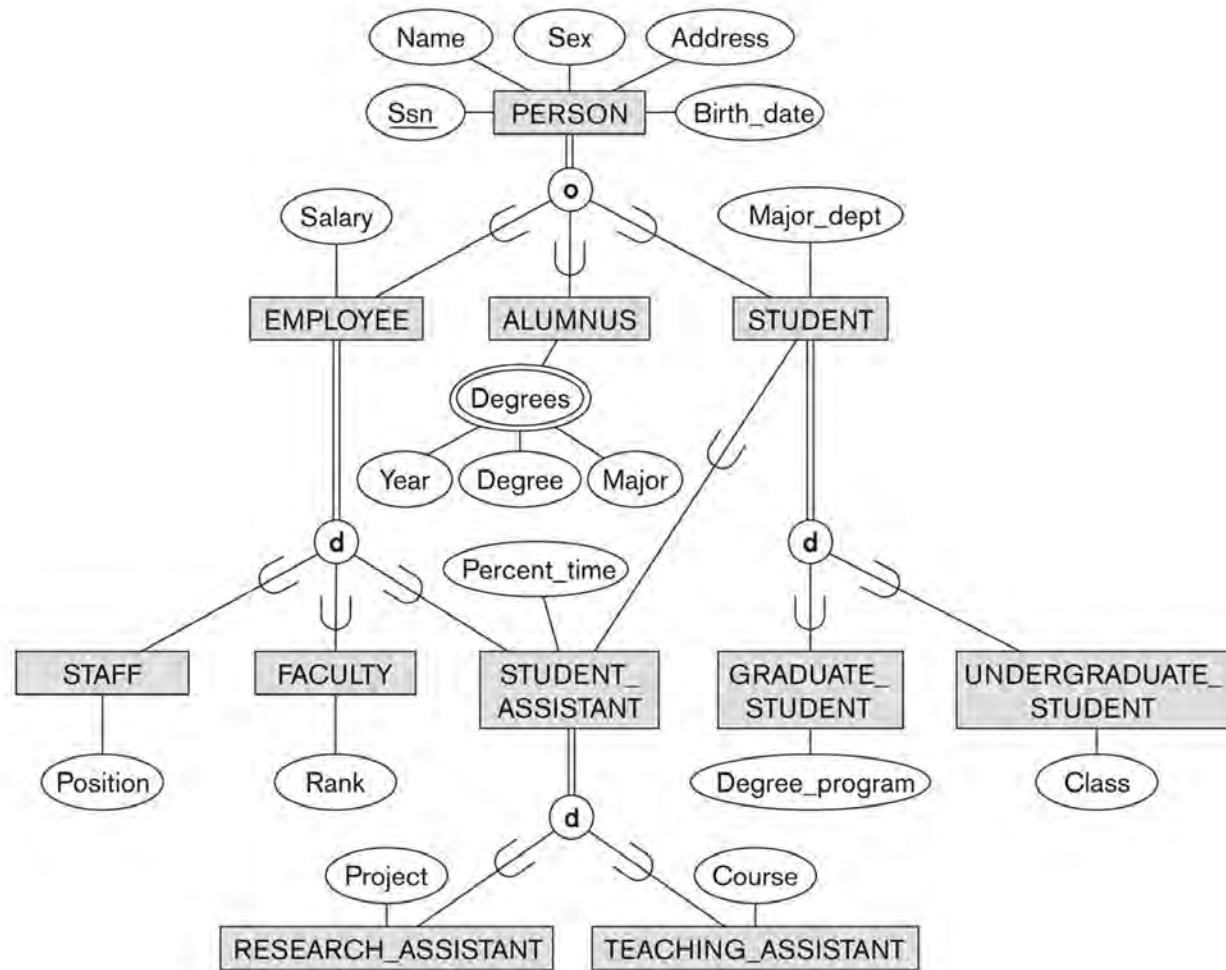
- **Hierarchy:** every subclass has only one superclass
 - realized by: **single inheritance**
 - forms a **tree structure**
- **Lattice:** a subclass can be subclass of more than one superclass
 - realized by: **multiple inheritance**
 - called: **shared subclasses**

Example – Hierarchies, Lattices



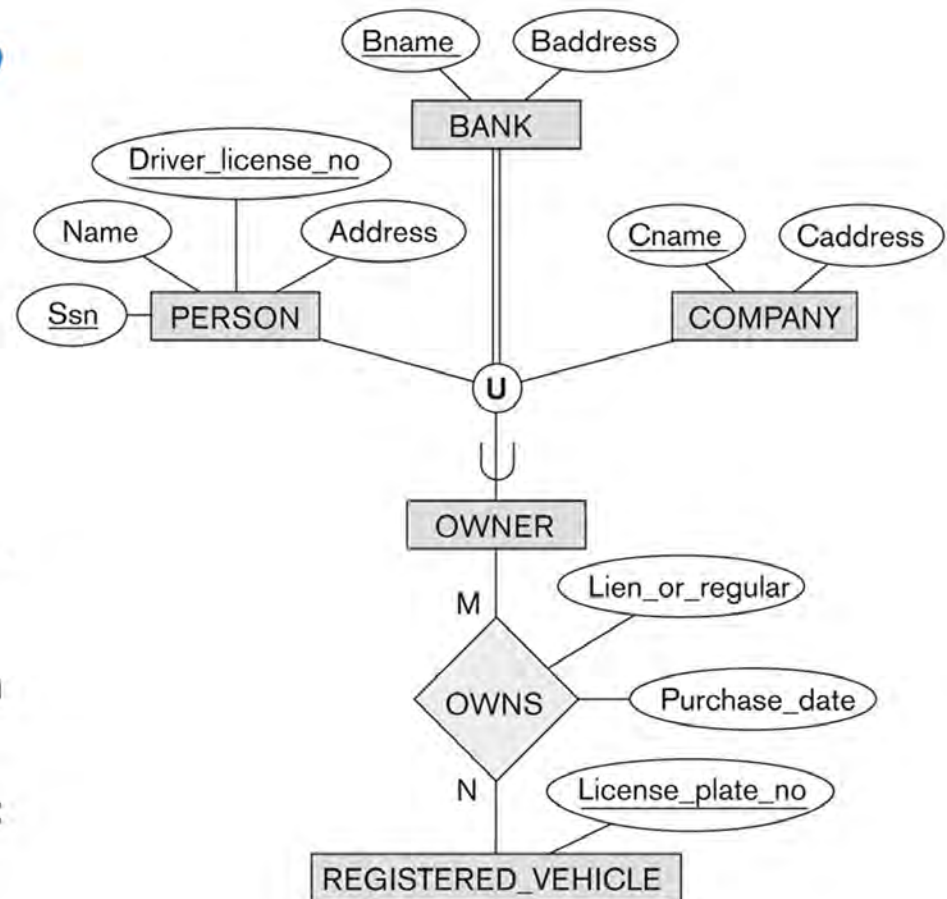
Shared subclass : ENGINEERING_MANAGER

Example – UNIVERSITY



Categories (UNION TYPES)

- **Union:** a single superclass/subclass *relationship* with multiple superclasses
 - superclasses can represent different entity types
 - notation: **U**
- **Union type (category):** a *subclass* of distinct superclass entity types
 - a category member must exist in **at least one** of its superclasses
 - Difference from *shared subclass*: shared subclass member must exist in **all** of its superclasses

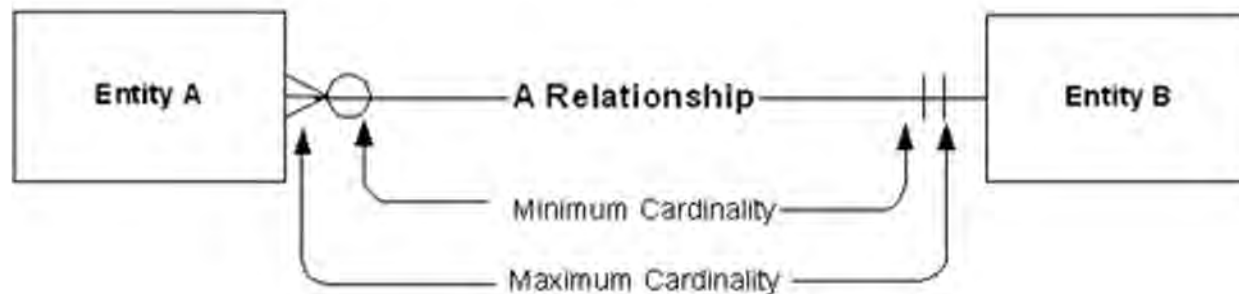


Alternative diagrammatic notations

Diagramming Conventions

- ER diagram is one popular example for displaying database schemas.
- Many other notations exist in the literature and in various database design and modeling tools.
- Example:
 - **Chen notation:** basic notation
 - **Crow's Foot Notation:** Crow's Foot diagrams represent entities as boxes, and relationships as lines between the boxes. Different shapes at the ends of these lines represent the cardinality of the relationship.
 - **UML:** UML Class Diagrams correspond to ER Diagram, but several differences exist.

Crow's Foot Notation



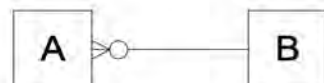
Each instance of A is related to a minimum of zero and a maximum of one instance of B



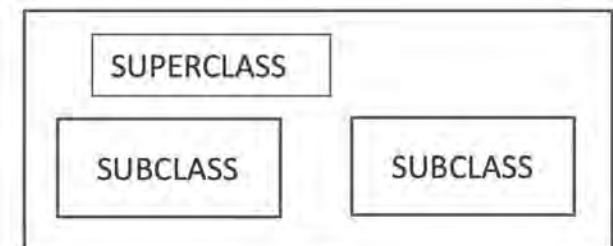
Each instance of B is related to a minimum of one and a maximum of one instance of A



Each instance of A is related to a minimum of one and a maximum of many instances of B



Each instance of B is related to a minimum of zero and a maximum of many instances of A



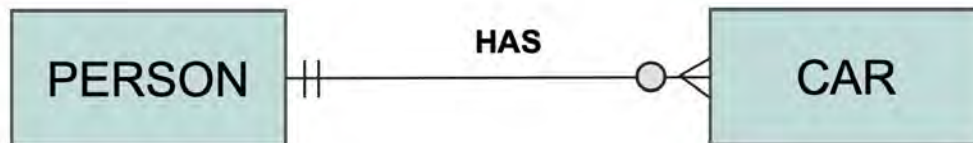
Equivalent Relationship Notations



Chen notation

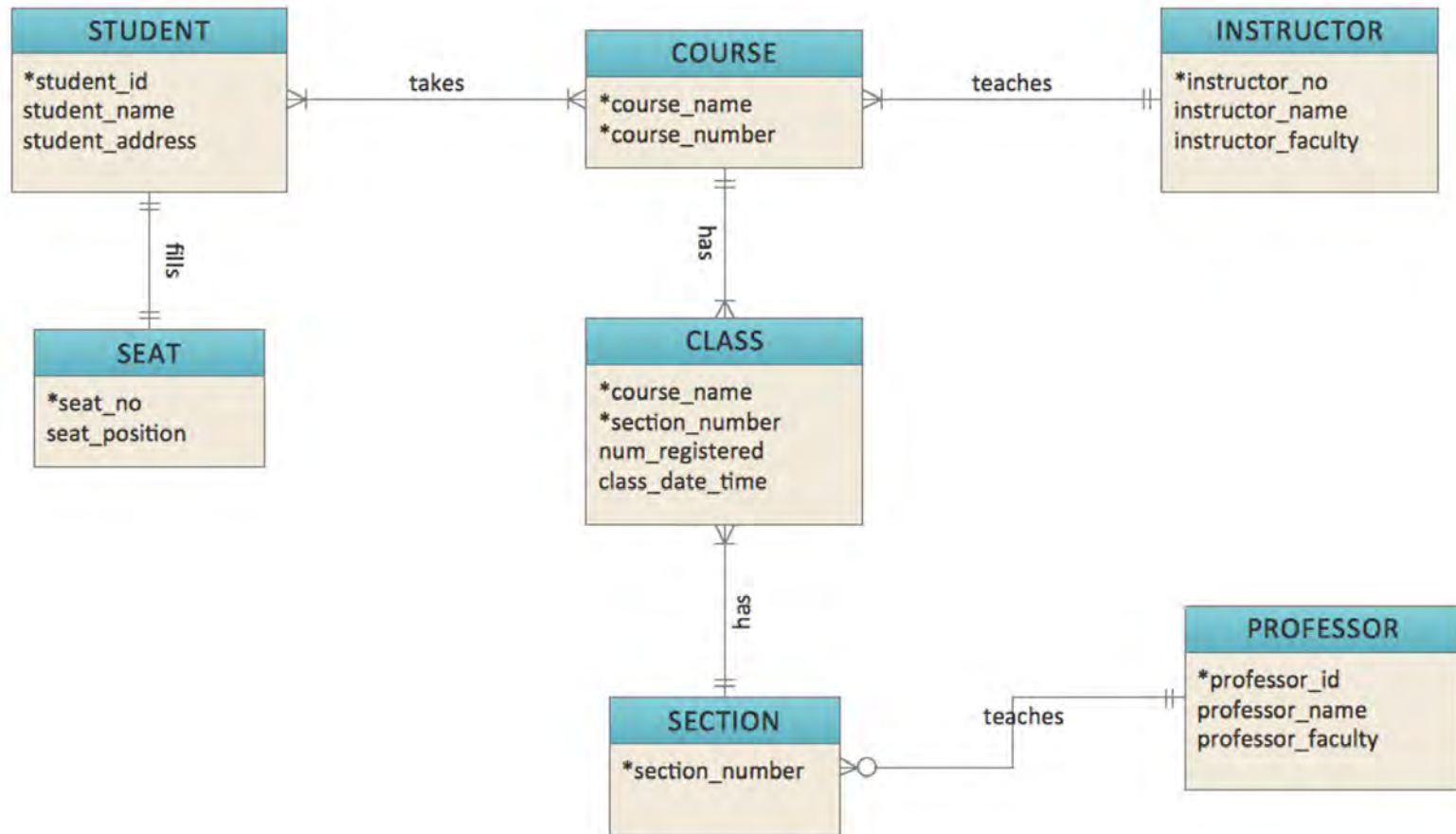


Min-Max notation

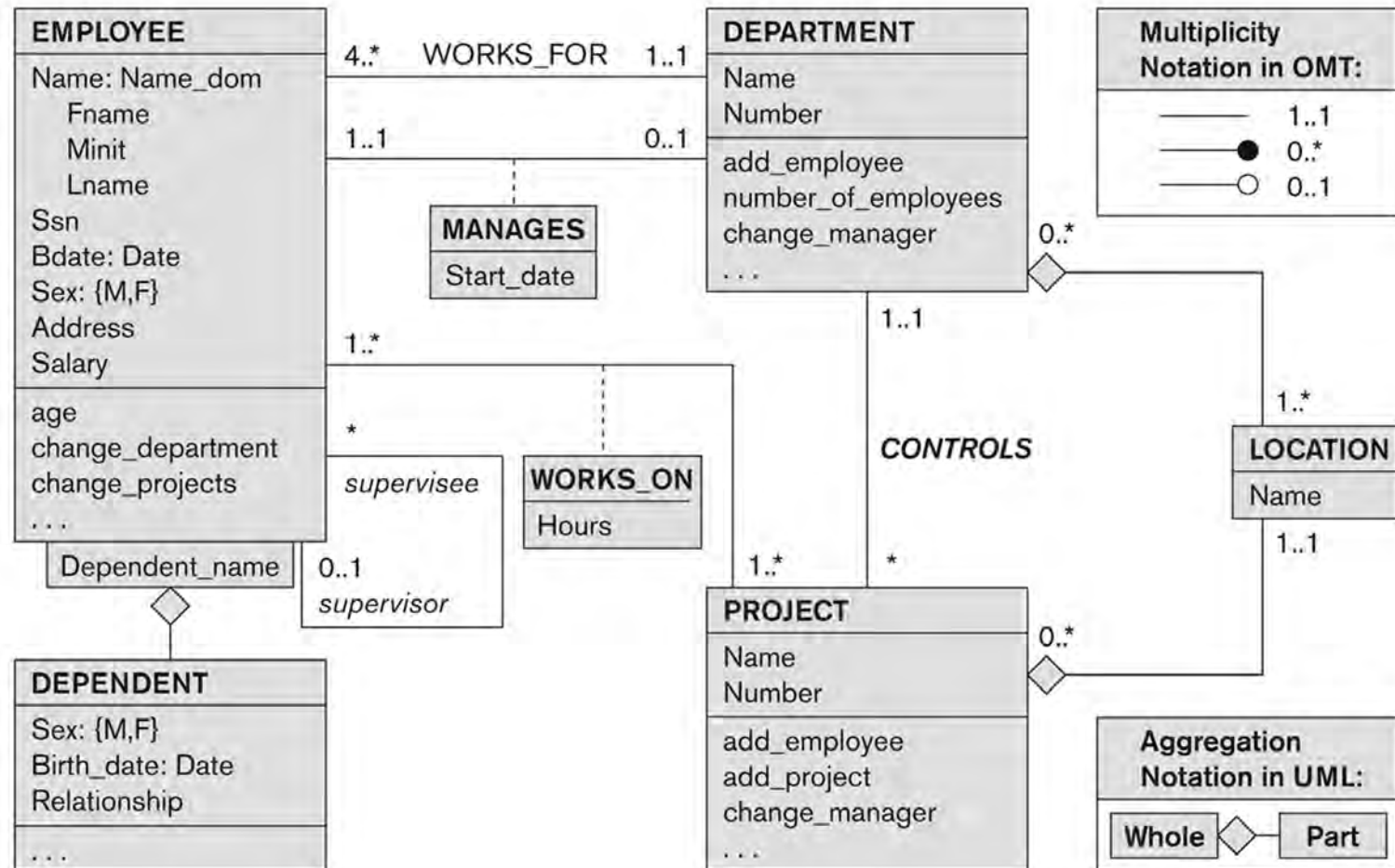


Crow's foot

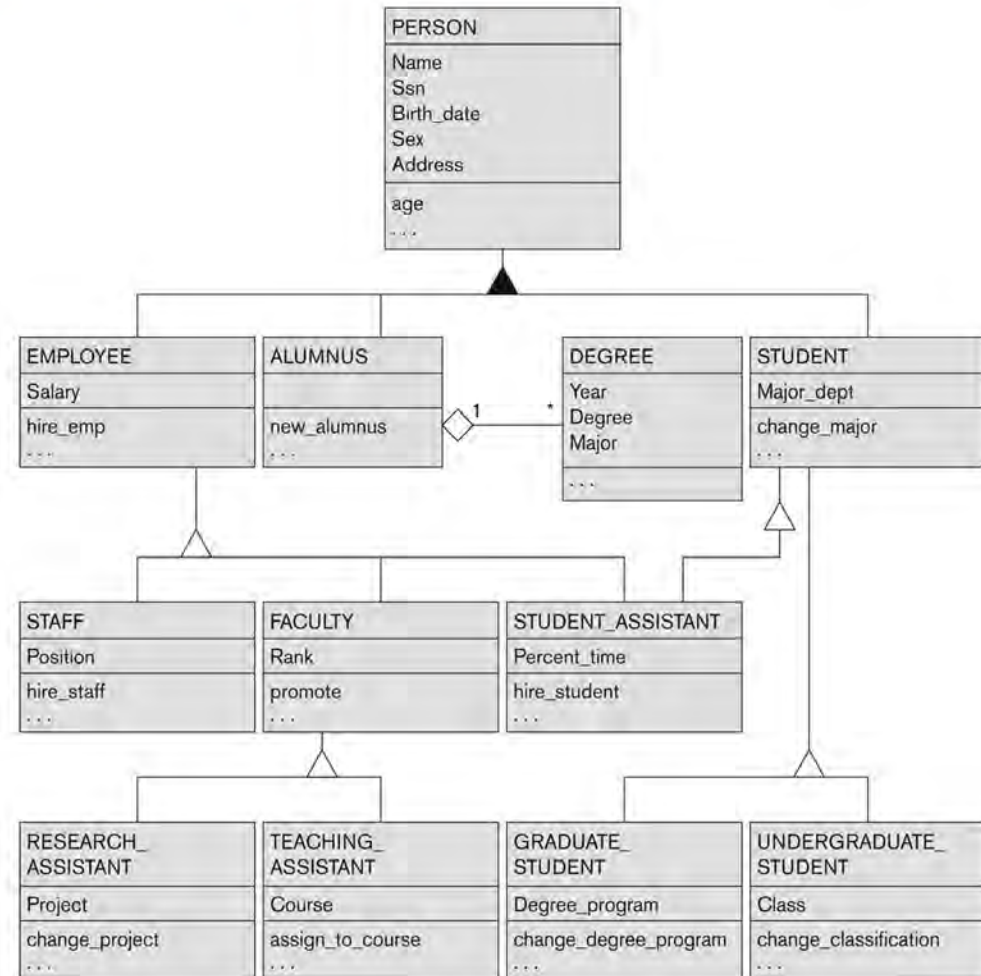
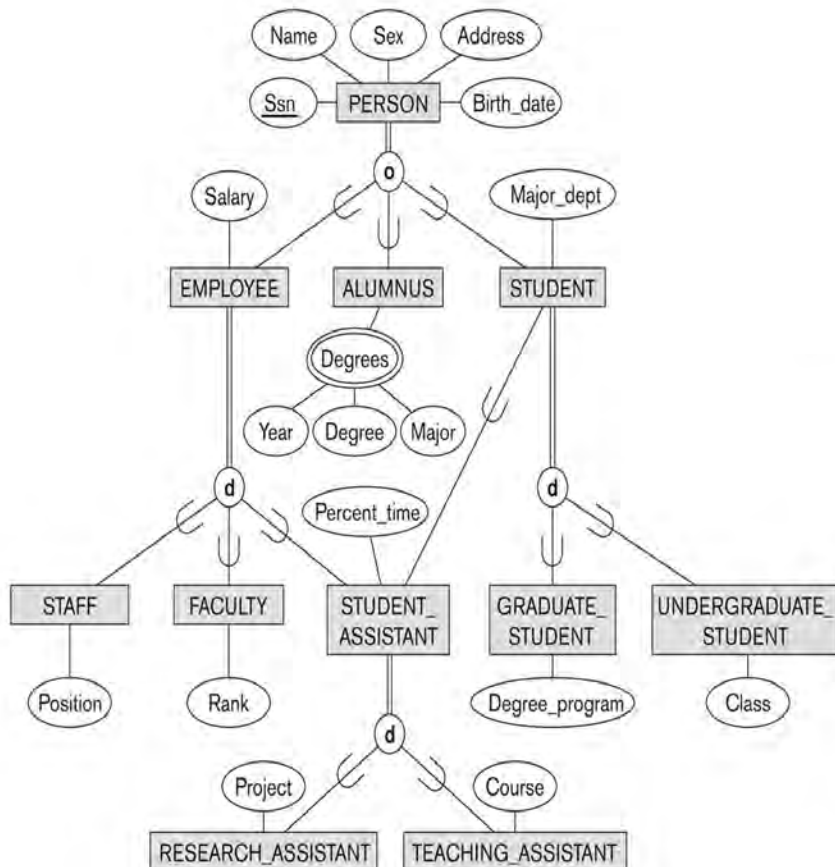
Example – Crow's Foot Notation



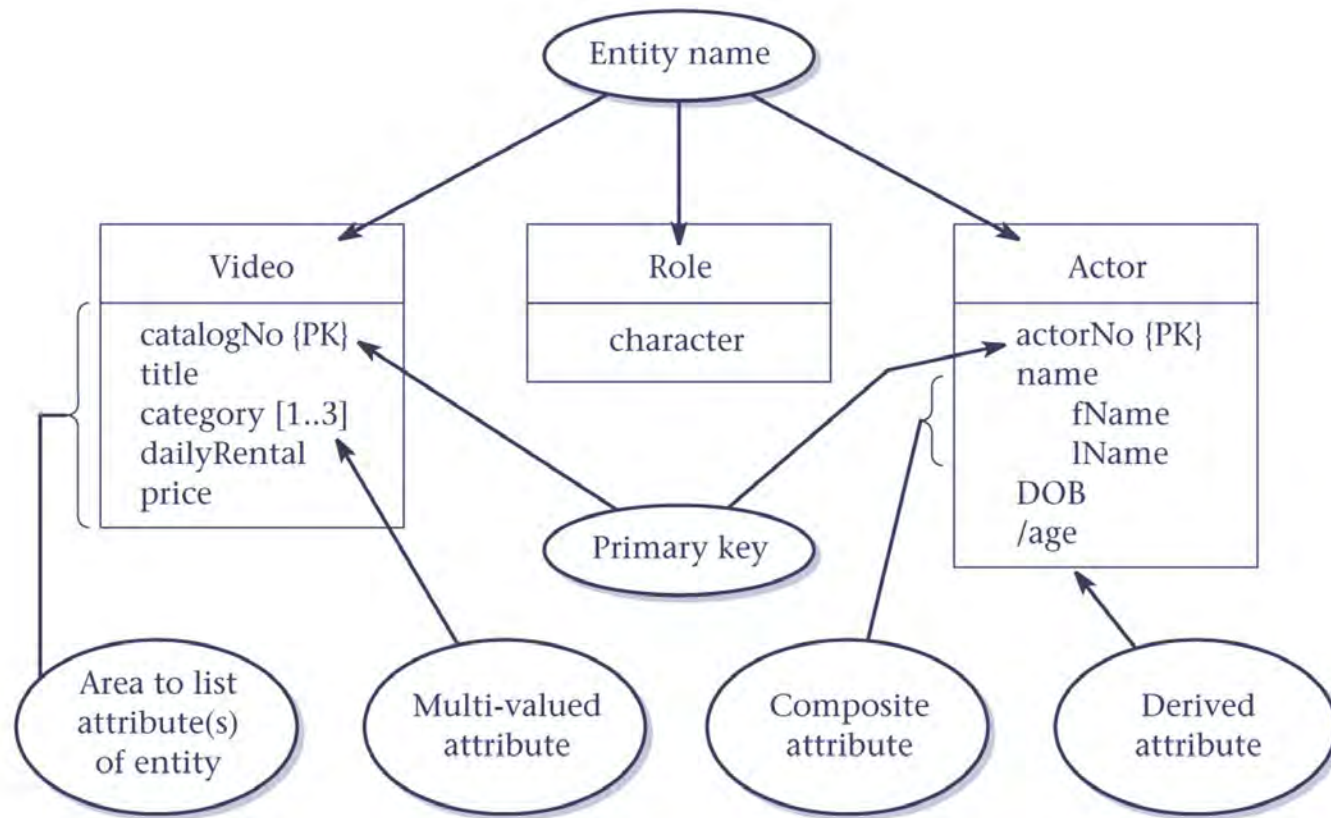
Example – UML Class Diagram for COMPANY



UML Example for Displaying Super/Subclasses



Other Alternative Diagrammatic Notations



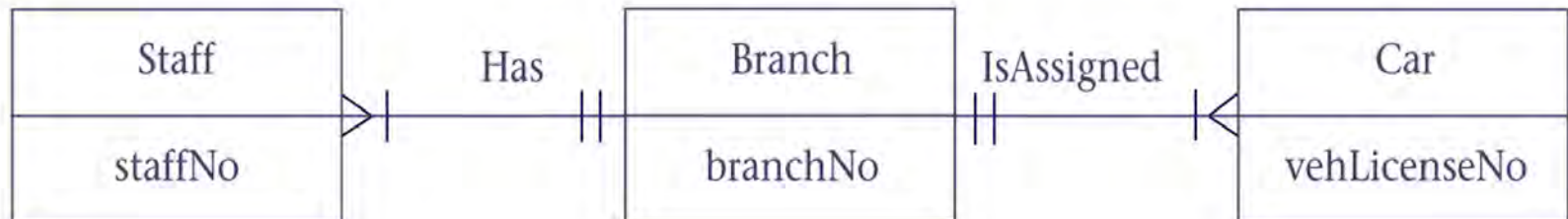
Problems with ER Model

Connection Traps

- **connection traps:** problems that may arise when relationships are not well-design in ER models
- 2 main types of connection traps:
 - fan traps
 - chasm traps

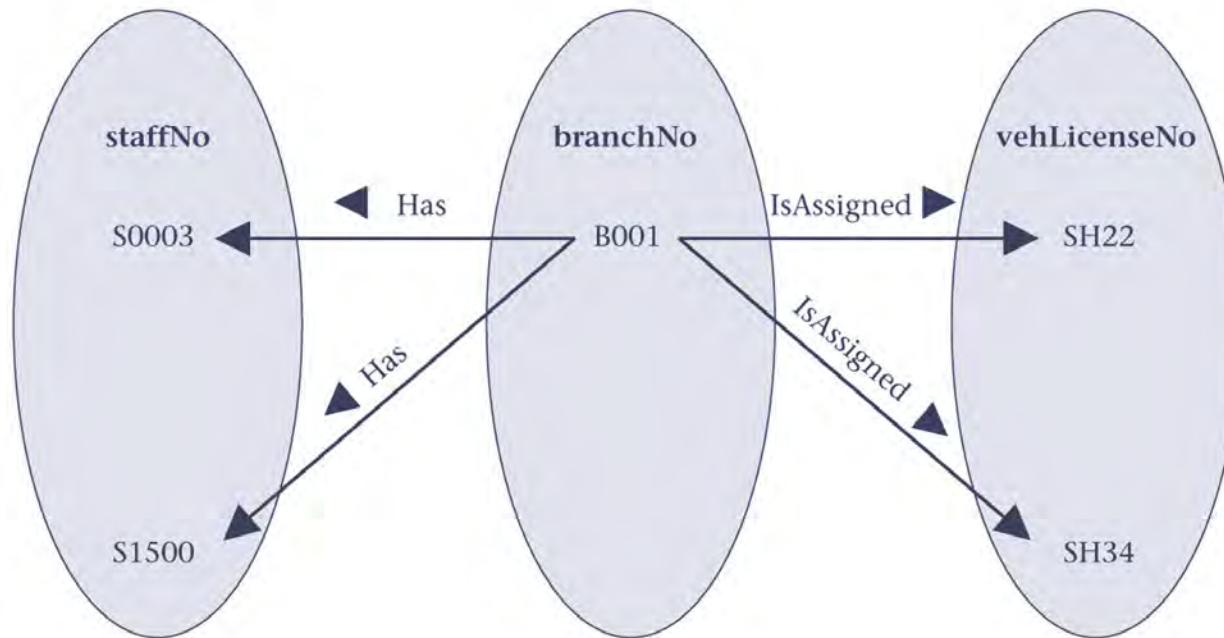
Fan Trap

- **Fan trap:** a model represents a relationship between entity types, but the *pathway* between certain entity occurrences *is ambiguous*.
 - It occurs when **1:N relationships** fan out from a single entity.
 - Example:

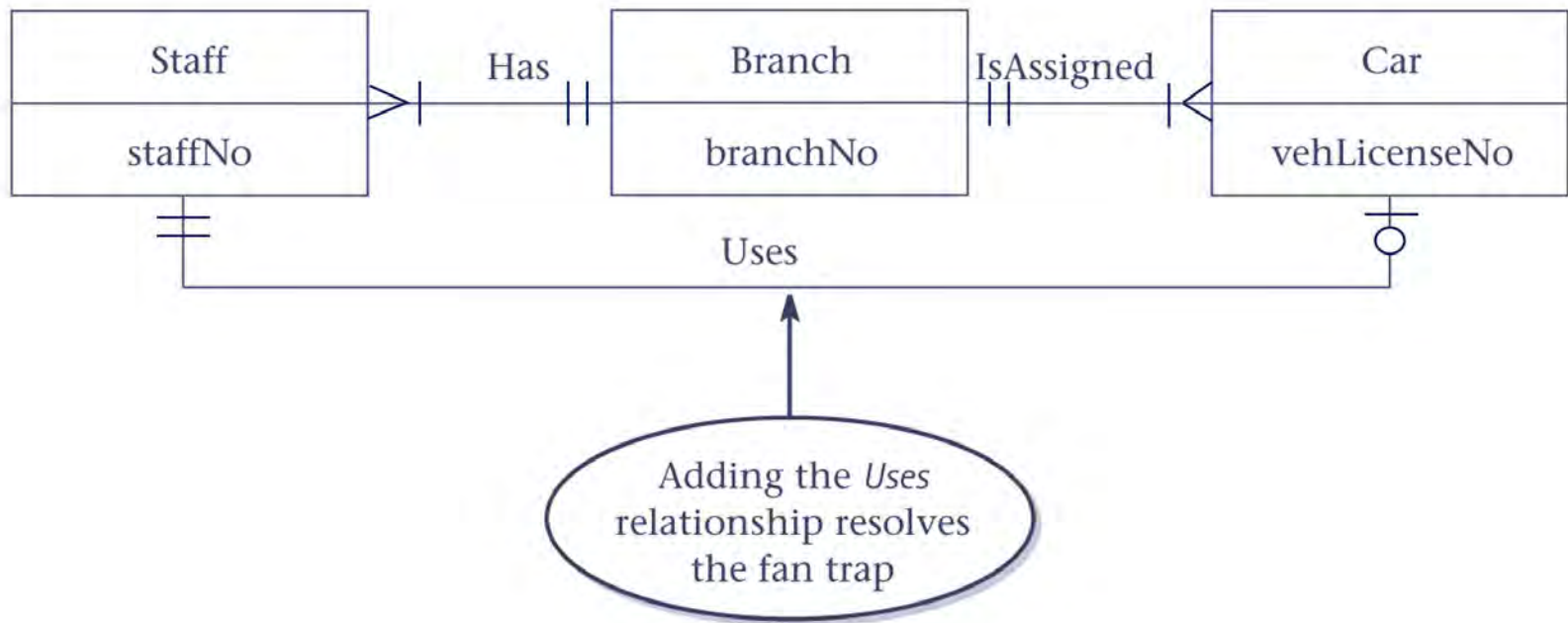


Example – Fan Trap

Cannot tell which member of staff uses car SH34.

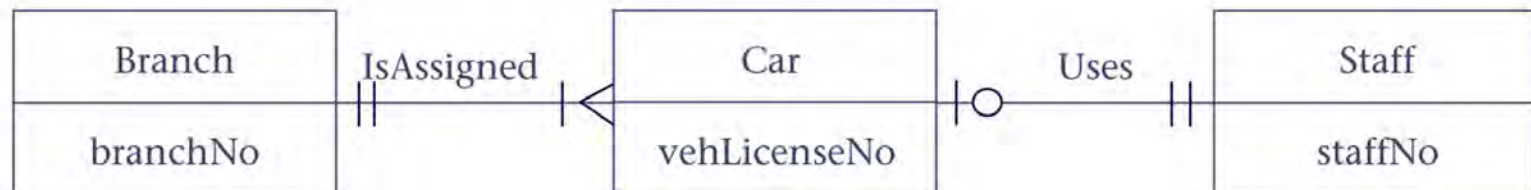


Resolving the Fan Trap



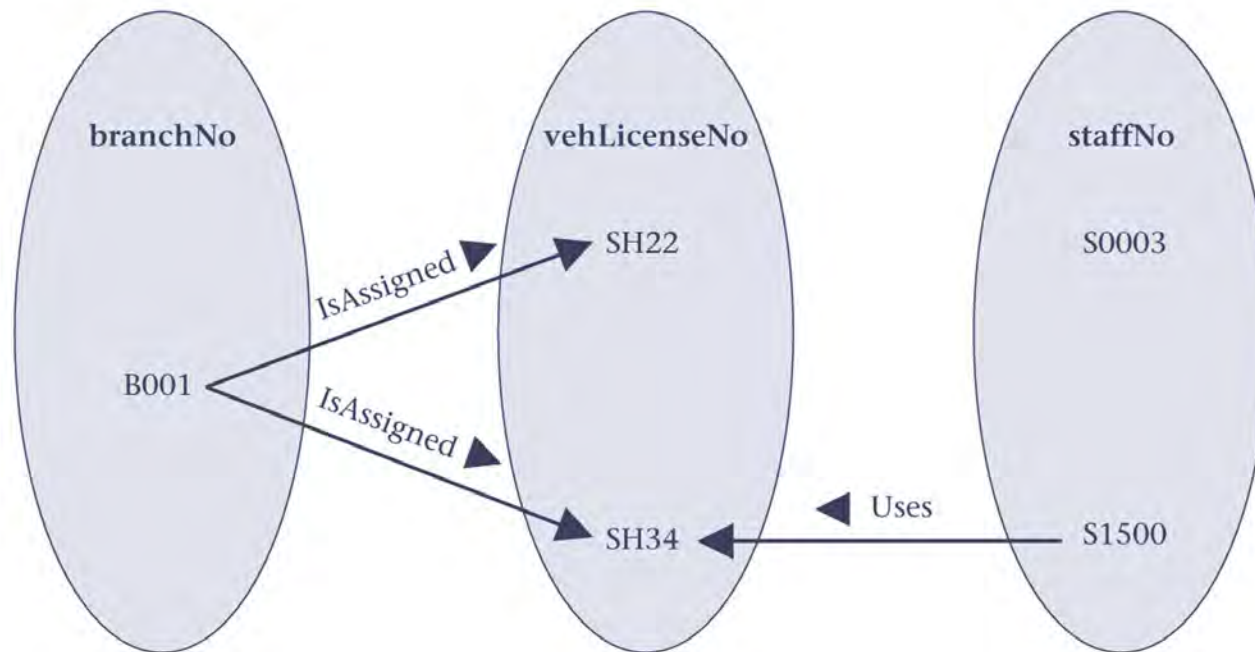
Chasm Trap

- **Chasm trap:** A model suggests the existence of a relationship between entities, but the *pathway does not exist* between certain entity occurrences.
 - It occurs where there is a **relationship with partial participation**, which forms part of the pathway between entities that are related.
 - Example:

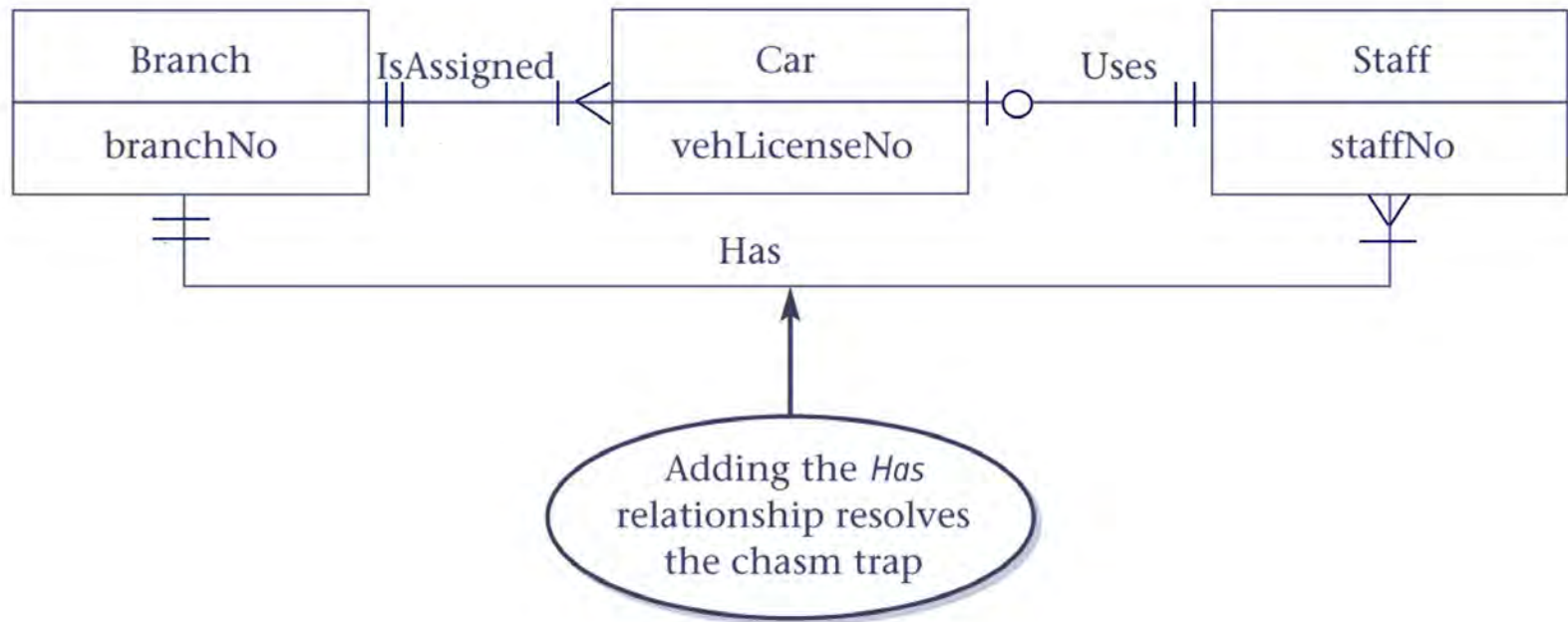


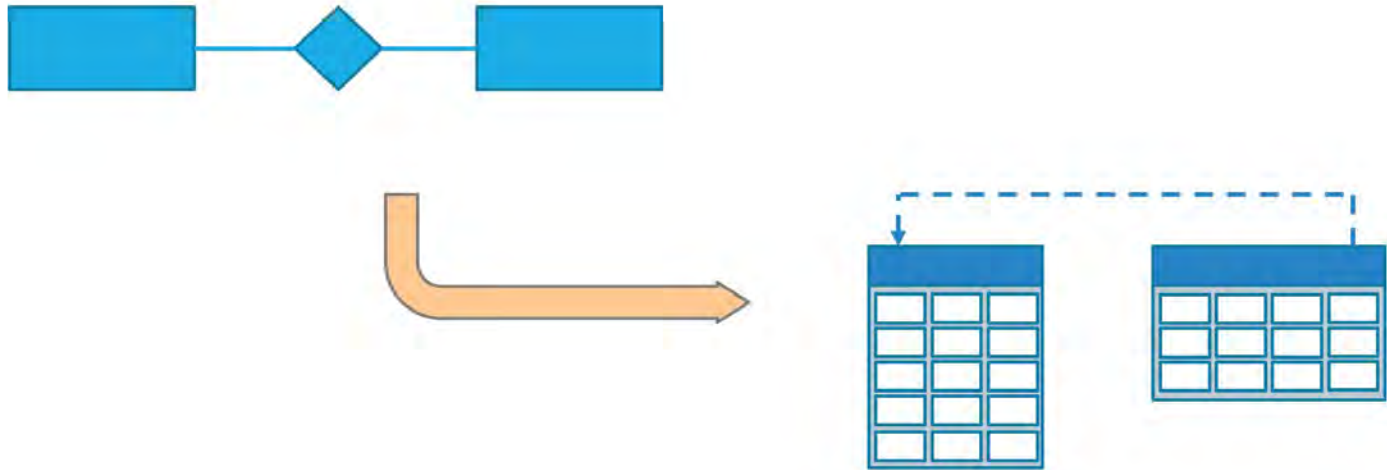
Example – Chasm Trap

Cannot tell which branch staff S0003 works at.



Resolving the Chasm Trap





Mapping:
(E)ER Model to Relational

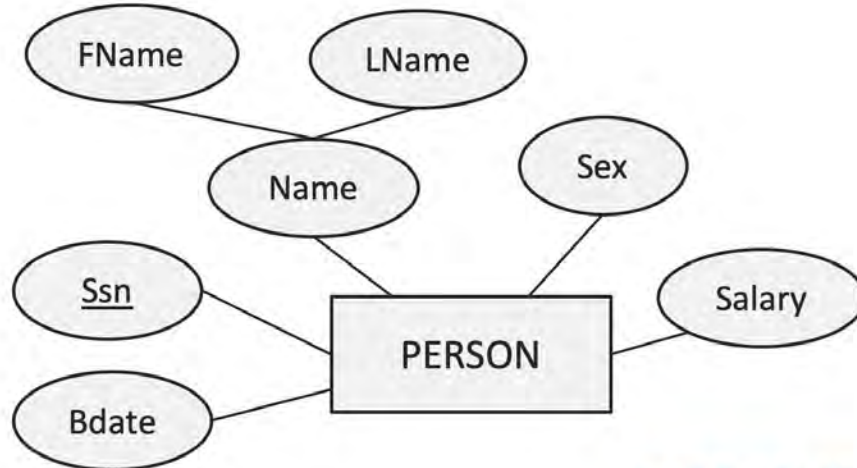
Table of Content

- Mapping ER model concepts
 - Mapping regular entity types, simple, composite and key attributes
 - Mapping multivalued attributes
 - Mapping weak entity types
 - Mapping binary M:N relationship types
 - Mapping binary 1:N relationship types
 - Mapping binary 1:1 relationship types
 - Mapping n -ary relationship types
 - Mapping recursive relationship types
- Mapping EER Model concepts
 - Mapping specialization or generalization
 - Mapping union types

Mapping Regular Entity Types, Simple, Composite and Key Attributes

For each regular (strong) entity type E in the ER schema

- create a relation R that:
 - includes **all the simple attributes** of E ,
 - includes **all the simple component attributes of composite attributes** (but the composite attribute(s) not),
 - choose the **key attribute of E as the primary key for R** .

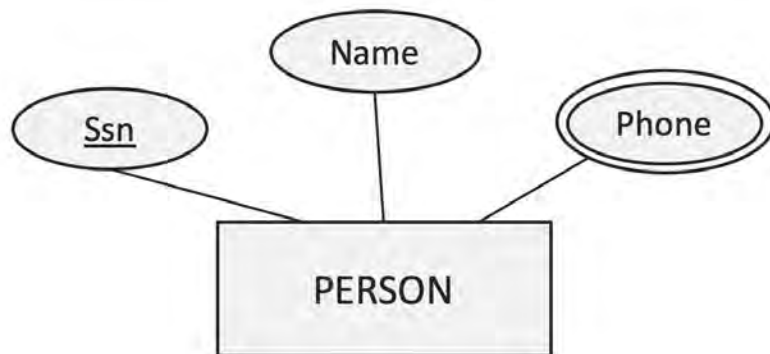


PERSON (Ssn, FName, Lname, Bdate, Sex, Salary)

Mapping Multivalued Attributes

For each multivalued attribute A ,

- create a new relation R , which
 - includes an attribute corresponding to A , plus the primary key attribute K – as a foreign key in R – of the relation that represents the entity type or relationship type that has A as an attribute.
 - The primary key of R is the combination of A and K .



PERSON(Ssn, Name)

PERSON_PHONE(Ssn, Phone)

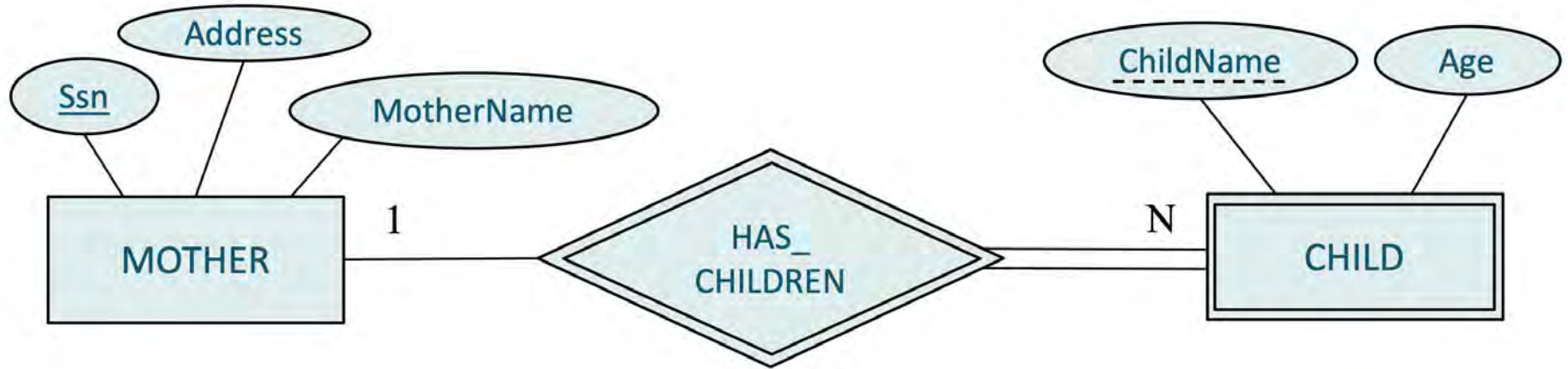


Mapping Weak Entity Types

For each weak entity type W with owner entity type E

- create a relation R , and
 - include all simple attributes (or simple components of composite attributes) of W as attributes of R ,
 - include as foreign key attribute(s) of R the primary key attribute(s) of the relation that correspond to the owner entity type.
- The primary key of R is the combination of the primary key of the owner and the partial key of the weak entity type W .

Mapping Weak Entity Type – Example



MOTHER (Ssn, MotherName, Address)

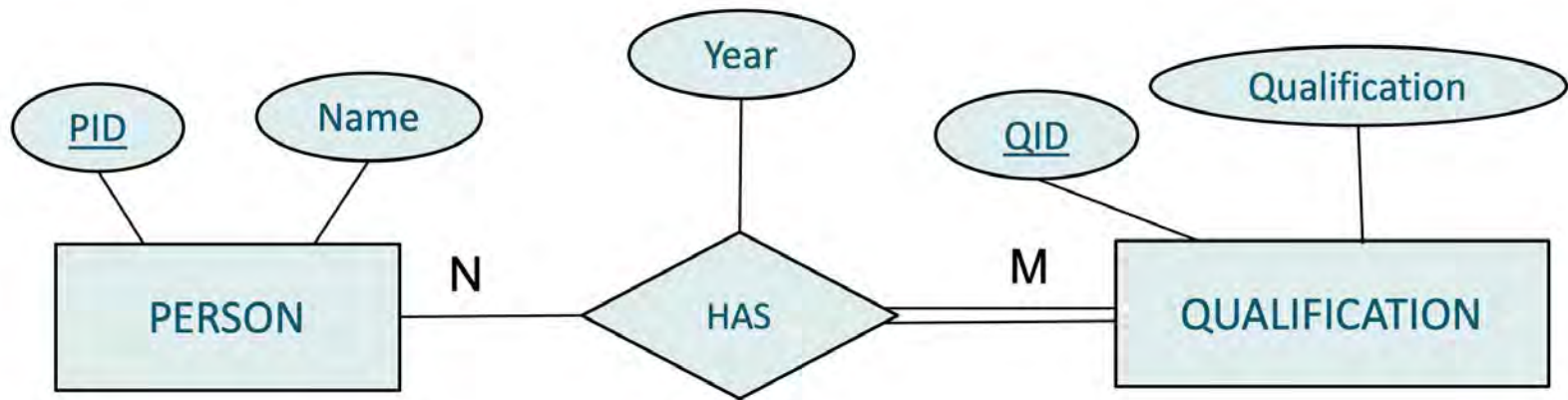
CHILD (Ssn, ChildName, Age)

Mapping Binary M:N Relationship Types

For each regular binary M:N relationship type R

- create a new relation S to represent R , and
 - include as foreign key attributes in S the primary keys of the relations that represent the participating entity types
 - combination of the key attributes of participating entity types will form the primary key of S
 - include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S .

Mapping Binary M:N Relationship Type – Example



PERSON (PID, Name)

QUALIFICATION (QID, Qualification)

PERSON_QUAL (PID, QID, Year)

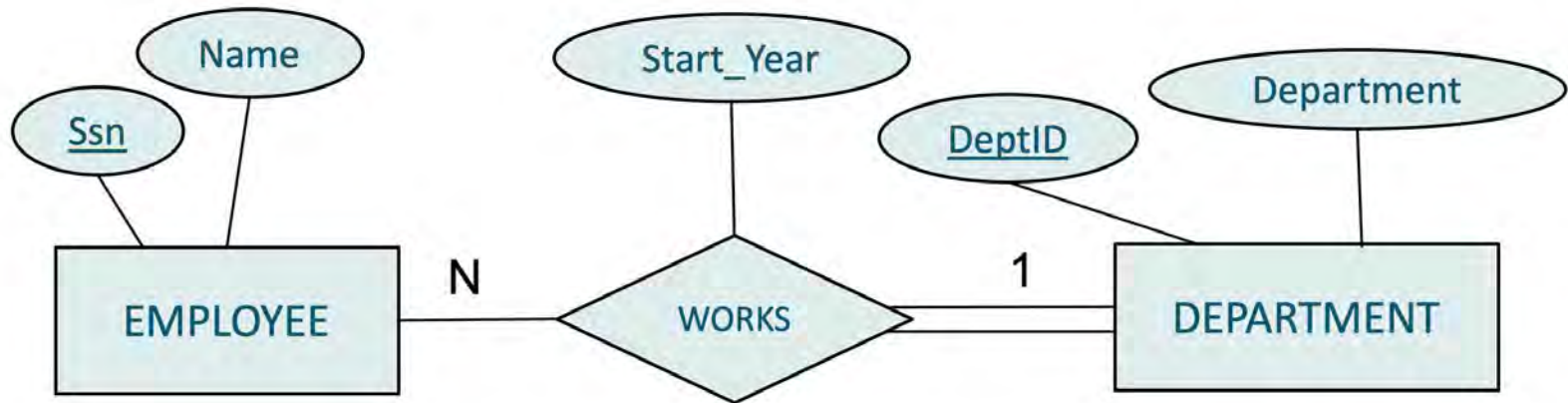
Mapping Binary 1:N Relationship Types

For each regular binary 1:N relationship type R ,

- identify the relation S that represents the participating entity type at the N -side of the relationship type,
- include as foreign key in S the primary key of the relation T that represents the other entity type (1-side) participating in R ,
- include any simple attributes (or simple components of composite attributes) of the 1:N relation type as attributes of S .

Note: When the N -side entity (S) participates in the relationship as a partial entity type, you may want to consider introducing a new relation (R). The primary key of R will be the primary key of S , the foreign keys are the key attributes of T and S . In this case, any simple attribute of the relationship type will also take a place in the new relation R .

Mapping Binary 1:N Relationship Type – Example



EMPLOYEE (Ssn, Name, *DeptID*, Start_Year)

DEPARTMENT (DeptID, Department)

Mapping Binary 1:1 Relationship Types

For the binary 1:1 relationship type R , identify the relations S and T that correspond to the entity types participating in R .

There are three possible mapping approaches:

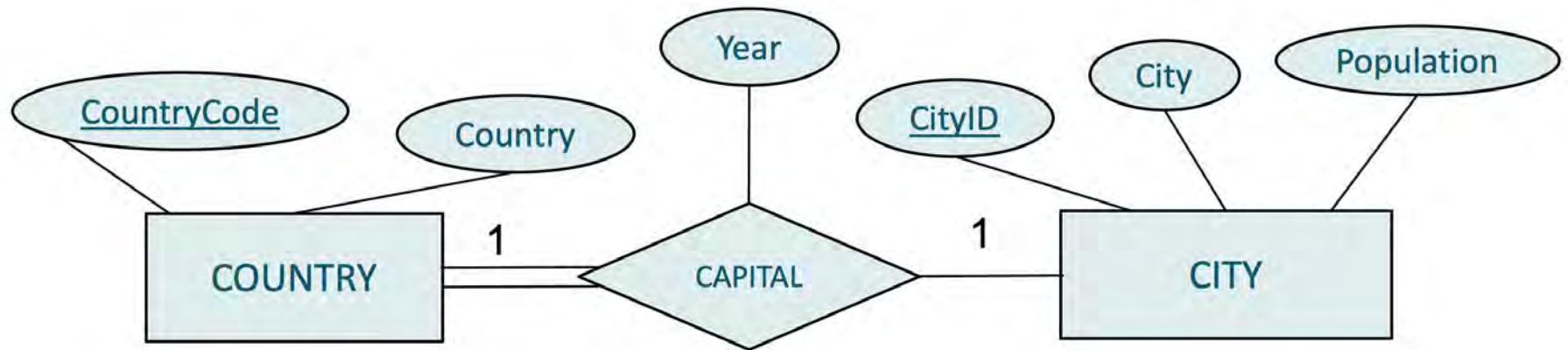
1. total-partial 1-1 relationship: **Foreign key approach**
2. total-total 1-1 relationship: **Merged relation option**
3. partial-partial 1-1 relationship: **Relationship relation approach**

Mapping Binary Total – Partial 1:1 Relationship Types

Foreign Key approach (total-partial 1-1):

- Choose one of the entity types – say S – and include as foreign key in S the key attribute of T .
 - It is better to choose an entity type with total participation in R in the role of S .
- If the relationship type has simple attributes, include them (or simple components of composite attributes) as attributes of S .

Foreign Key Approach for 1:1 Relationship – Example



COUNTRY (CountryCode, Country, *CapitalCityID*, Year)

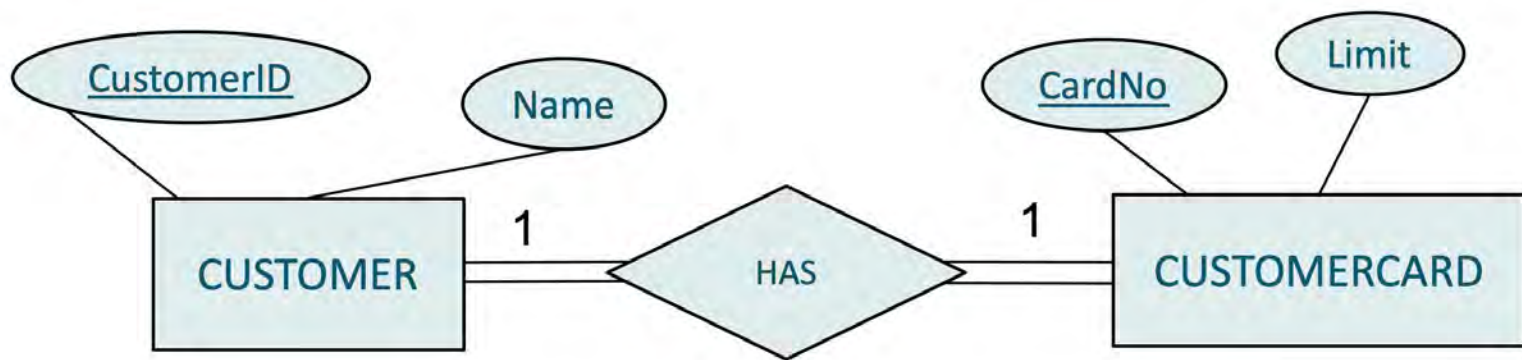
CITY (CityID, City, Population)

Mapping Binary Total – Total 1:1 Relationship Types

Merged relation option (total-total 1-1):

- Merge the two entity types and the relationship into a single relation.
 - The primary key of the new merged relation is one of the primary keys of the participant relations.
 - It is suggested to use the primary key of the more important entity type.
 - If there are any attributes in common, the duplicates are removed.
 - If the relationship type has simple attributes, include them (or simple components of composite attributes) as attributes in the merged relation.

Merged Relation Option for 1:1 Relationship – Example



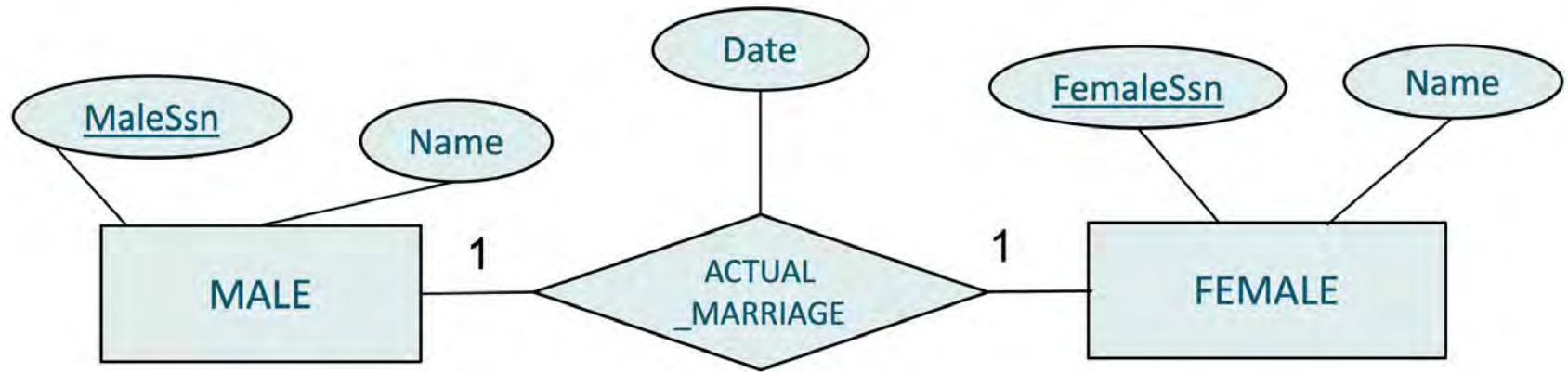
CUSTOMER (CustomerID, Name, CardNo, Limit)

Mapping Binary Partial – Partial 1:1 Relationship Types

Relationship relation approach (partial-partial 1-1):

- Set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T .
 - Arbitrary select one of the primary keys of S and T to be the primary key of R .
 - In the new relation R the primary keys of T and S are both foreign keys which refer to the own relation mapped from the entity type.
 - If the relationship type has simple attributes, include them (or simple components of composite attributes) as attributes in the new relation R .

Relationship Relation Approach for 1:1 Relationship – Example



MALE (MaleSsn, Name)

FEMALE (FemaleSsn, Name)

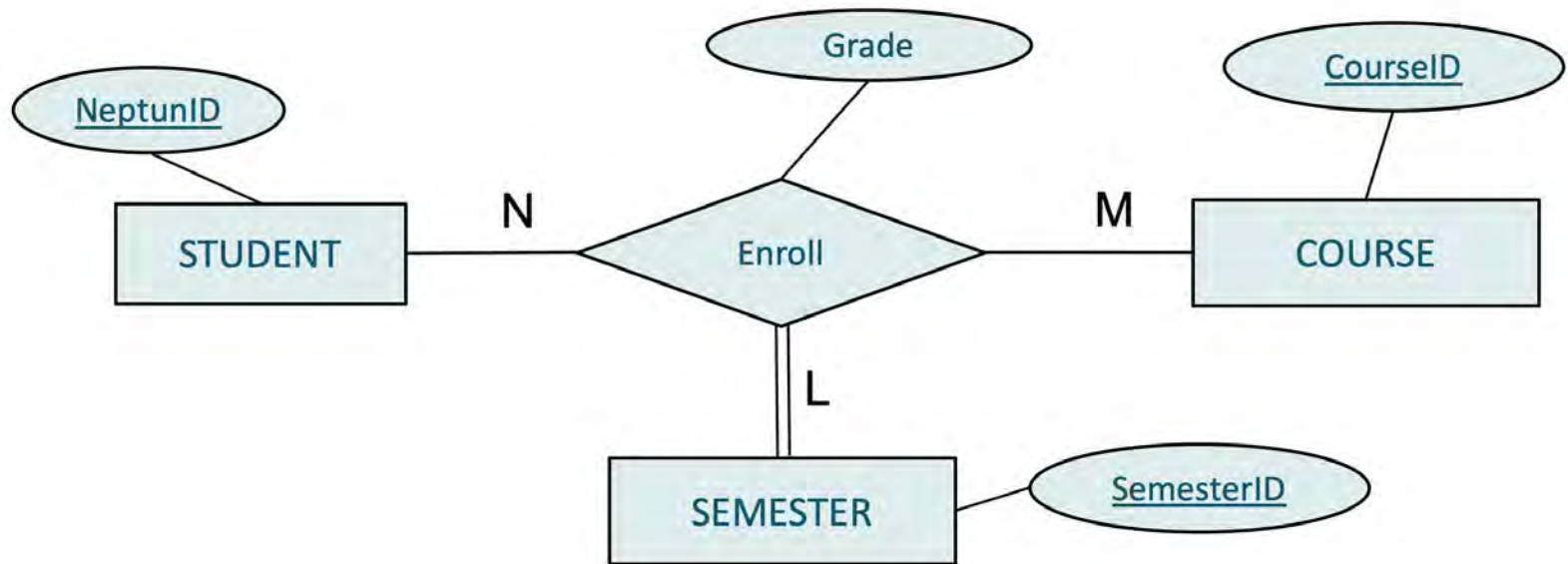
ACTUAL_MARRIAGE (MaleSsn, FemaleSsn, Date)

Mapping n -ary Relationship Types

For each n -ary relationship type R , where $n > 2$,

- create a new relation S to represent R , and
- include as foreign key attributes in S the primary keys of the relations that represent the participating entity types.
- The primary key for S is the combination of the of primary keys of entity types representing the **N -side** participating entity types. When the n -ary relationship connects only „1-side” entity types, then the primary key of the S relation is one of the primary keys of the relations representing the participating entity types (arbitrarily chosen).
- Also include any simple attributes of the n -ary relationship type (or simple components of composite attributes) as attributes of S .

Mapping n -ary Relationship Type – Example



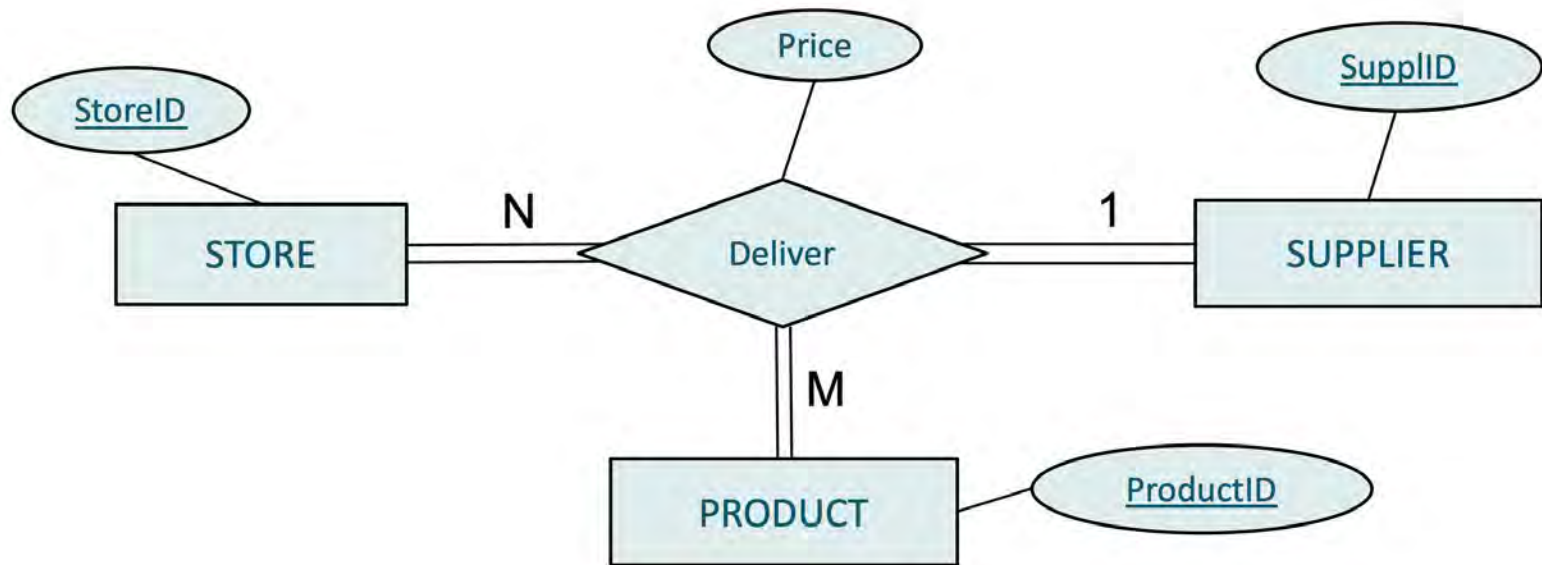
STUDENT (NeptunID, ...)

COURSE (CourseID, ...)

ENROLL(NeptunID, CourseID, SemesterID, Grade)

SEMESTER (SemesterID, ...)

Mapping n -ary Relationship Type – Example



STORE (StoreID, ...)

SUPPLIER (SupplID, ...)

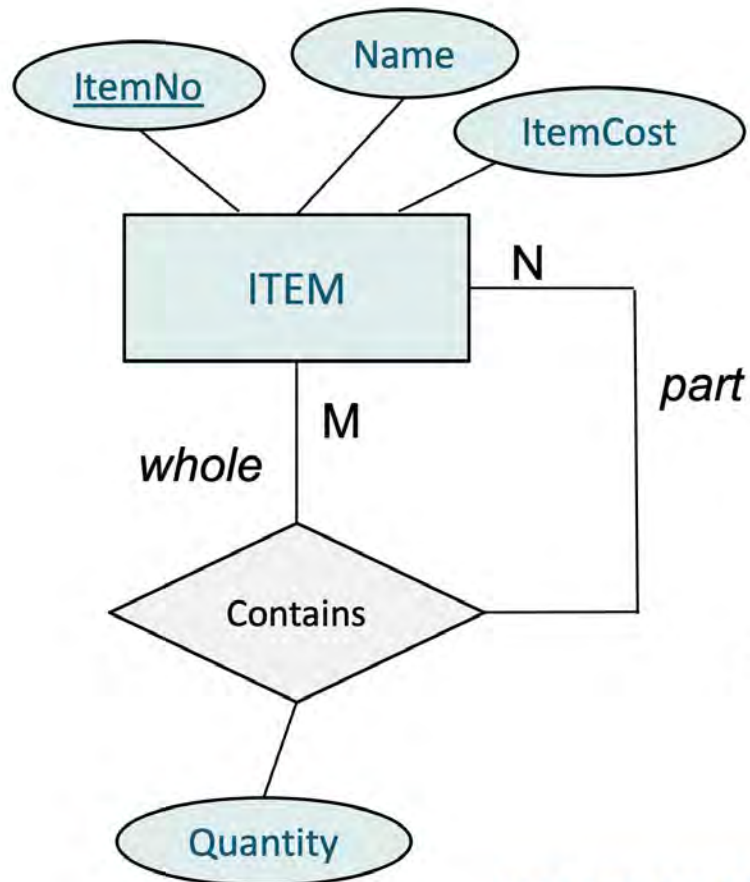
DELIVER(StoreID, ProductID, SupplID, Price)

PRODUCT (ProductID, ...)

Mapping Recursive Relationship Types

- Use the same mapping rules as it was shown for mapping 1:1, 1:N and M:N relationship types.
 - Rename the relations and the attributes if it needed.

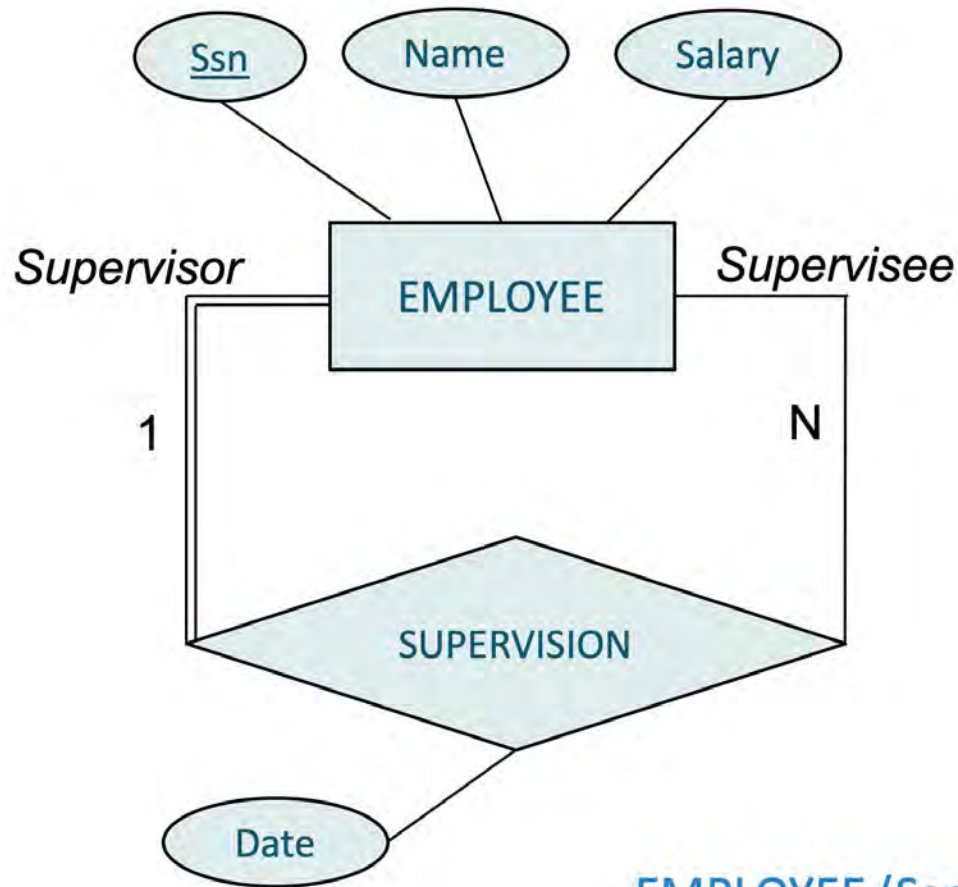
Mapping Recursive Relationship Type – Example 1 (M:N)



ITEM (ItemNo, Name, ItemCost)

COMPONENT (MainItemNo, ComponentNo, Quantity)

Mapping Recursive Relationship Type – Example 2 (1:N)



EMPLOYEE (Ssn, Name, Salary, *BossSsn*, Date)

Mapping Specialization or Generalization

4 possible options for mapping specialization or generalization:

- Multiple relations: superclass and subclasses
- Multiple relations: subclass relations only
- Single relation with one type attribute
- Single relation with multiple type attributes

Multiple Relations: Superclass and Subclasses

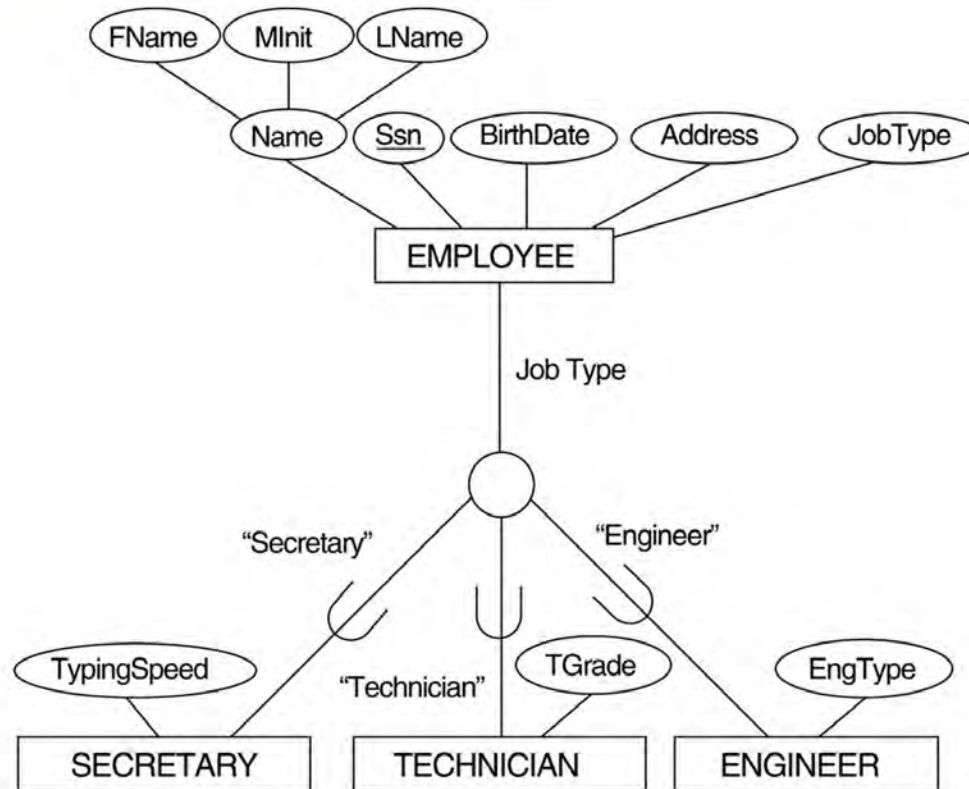
Let C the superclass with attributes of $\{k, a_1, \dots, a_n\}$, where k is the primary key of C , and let S_1, S_2 and S_m the subclasses of C .

Multiple relations: superclass and subclasses

- Create a relation L for C with attributes
 $Attrs(L) = \{k, a_1, \dots, a_n\}$ and $PK(L) = k$, and
- create a relation L_i for each subclass $S_i, 1 < i < m$, with the attributes
 $Attrs(L_i) = \{k\} \cup \{attributes\ of\ S_i\}$ and $PK(L_i) = k$.
- In each L_i relation: $FK(L_i) = k$, that refers to the relation L .
- This option works for any specialization (total or partial, disjoint or overlapping).

Multiple Relations: Superclass and Subclasses

– Example



EMPLOYEE (Ssn, FName, MInit, LName, BirthDate, Address)

SECRETARY (Ssn, TypingSpeed)

TECHNICIAN (Ssn, TGrade)

ENGINEER (Ssn, EngType)

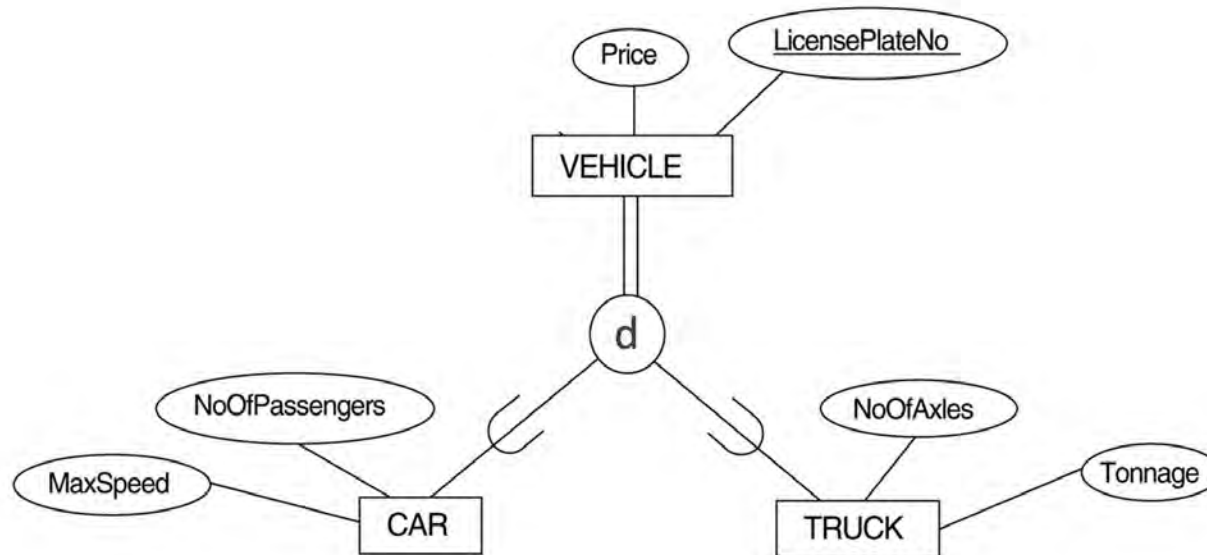
Multiple Relations: Subclass Relations Only

Let C the superclass with attributes of $\{k, a_1, \dots, a_n\}$, where k is the primary key of C , and let S_1, S_2 and S_m the subclasses of C .

Multiple relations: subclass relations only

- Create a relation L_i for each subclass $S_i, 1 < i < m$, with the attributes $Attr(L_i) = \{attributes\ of\ S_i\} \cup \{k, a_1, \dots, a_n\}$ and $PK(L_i) = k$.
- This option works well only for disjoint and total constraints.

Multiple Relations: Subclass Relations only – Example



CAR (LicensePlateNo, Price, MaxSpeed, NoOfPassengers)

TRUCK (LicensePlateNo, Price, NoOfAxles, Tonnage)

Single Relation with one Type Attribute

Let C the superclass with attributes of $\{k, a_1, \dots, a_n\}$, where k is the primary key of C , and let S_1, S_2 and S_m the subclasses of C .

Single relation with one type attribute

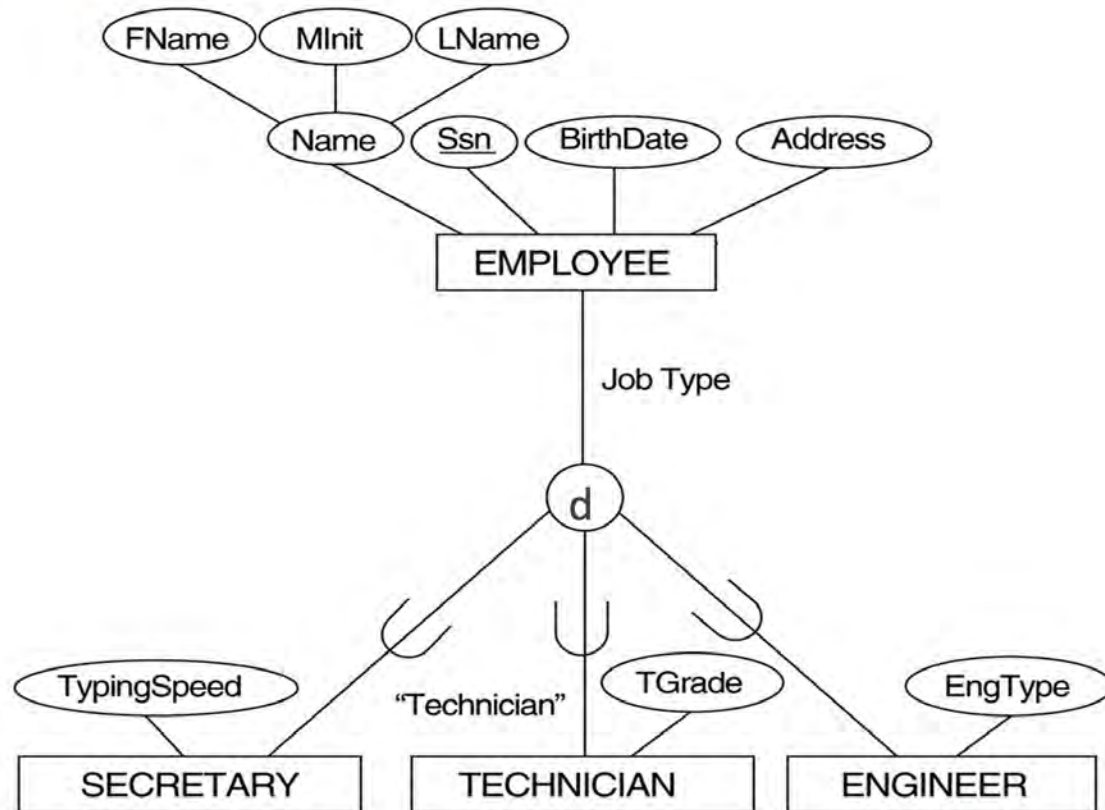
- Create a single relation L with attributes
 $Attrs(L) = \{k, a_1, \dots, a_n\} \cup$
 $\{attributes\ of\ S_1\} \cup \dots \cup \{attributes\ of\ S_m\} \cup \{t\}$
and $PK(L) = k$.

The attribute t is called a **type** (or **discriminating**) **attribute** that indicates the subclass to which the tuple belongs.

- This option is for specialization whose subclasses are DISJOINT.

Single Relation with one Type Attribute

– Example



EMPLOYEE (Ssn, FName, MInit, LName, BirthDate, Address, **JobType**, TypingSpeed, TGrade, EngType)

Single Relation with Multiple Type Attributes

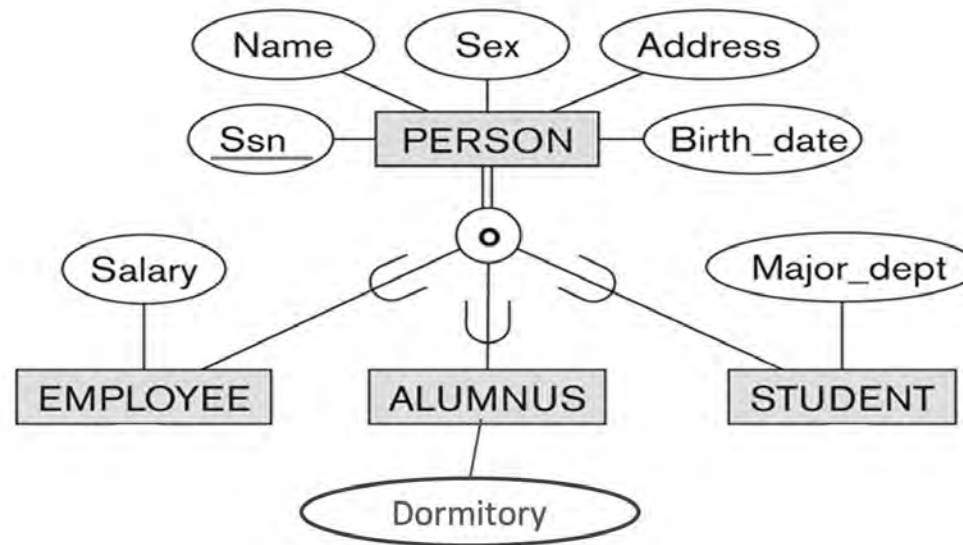
Let C the superclass with attributes of $\{k, a_1, \dots, a_n\}$, where k is the primary key of C , and let S_1, S_2 and S_m the subclasses of C .

Single relation with multiple type attributes

- Create a single relation L with attributes
$$Attrs(L) = \{k, a_1, \dots, a_n\} \cup \{attributes\ of\ S_1\} \cup \dots \cup \{attributes\ of\ S_m\} \cup \{t_1, t_2, \dots, t_m\}$$
and $PK(L) = k$.
- Each $t_i, 1 < i < m$, is a Boolean type attribute indicating whether a tuple belongs to the subclass S_i .
- This option works well for specialization whose subclasses are OVERLAPPING, but it could be used for disjoint subclasses too.

Single Relation with Multiple Type Attributes

– Example

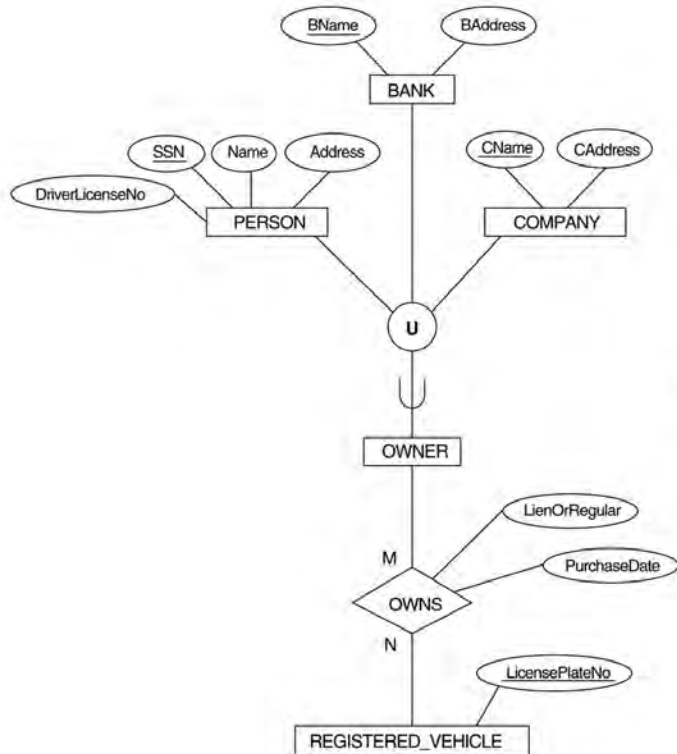


PERSON (Ssn, Name, Sex, Address, Birth_date, **EmployeeFlag**, Salary, **AlumnusFlag**, Dormitory, **StudentFlag**, Major_dept)

Mapping Union Types (Categories)

- Specify a new key attribute, called a **surrogate key**.
- The surrogate key allows to reference to the category.

Mapping Union Types – Example



PERSON (Ssn, DriverLicenseNo,
Name, Address, *OwnerID*)

BANK (BName, BAddress, *OwnerID*)

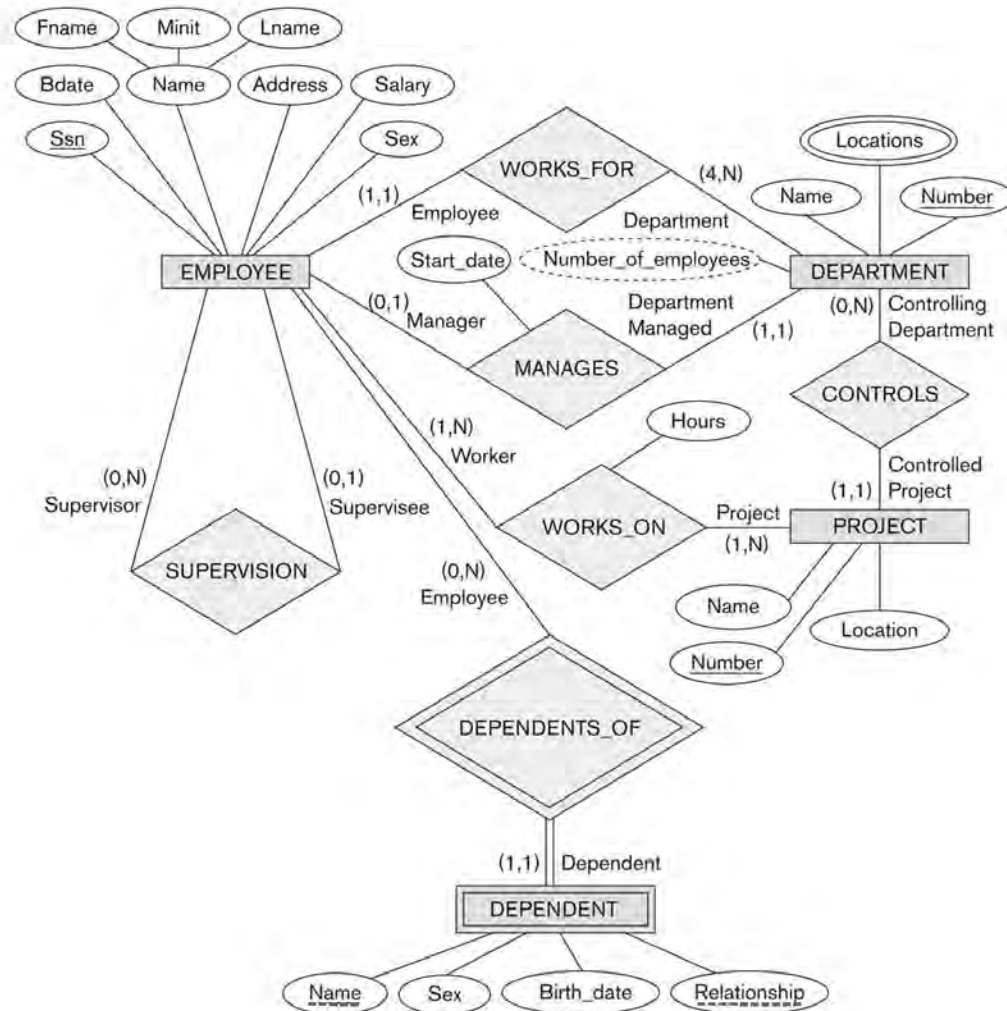
COMPANY (CName, CAddress, *OwnerID*)

OWNER (OwnerID)

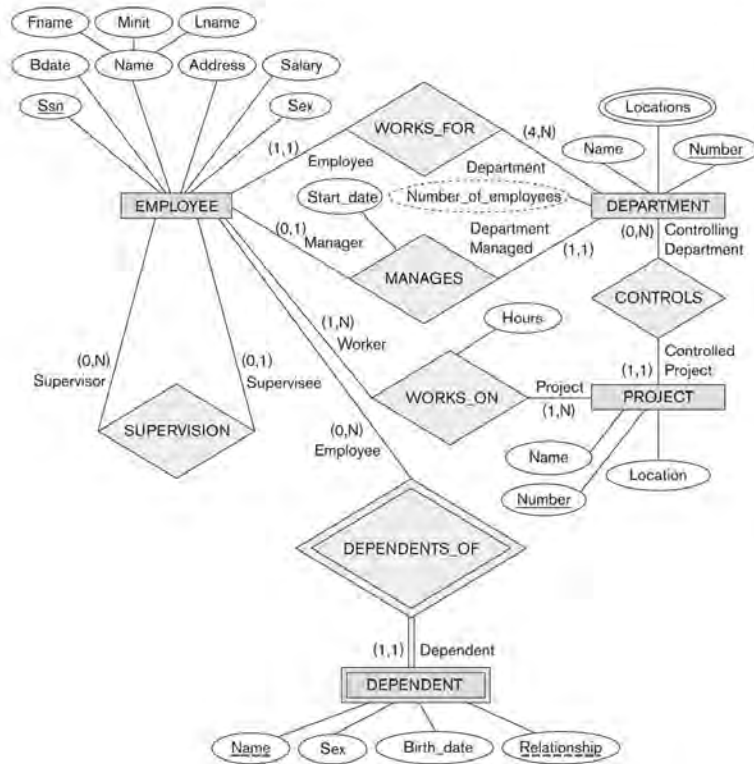
REGISTERED_VEHICLE (LicensePlateNo)

OWNS (OwnerID, LicensePlateNo, LienOrRegular, PurchaseDate)

Mapping Exercise



Mapping Exercise



DEPENDENT(Ssn, Name, Relationship, Sex, Birth_date)

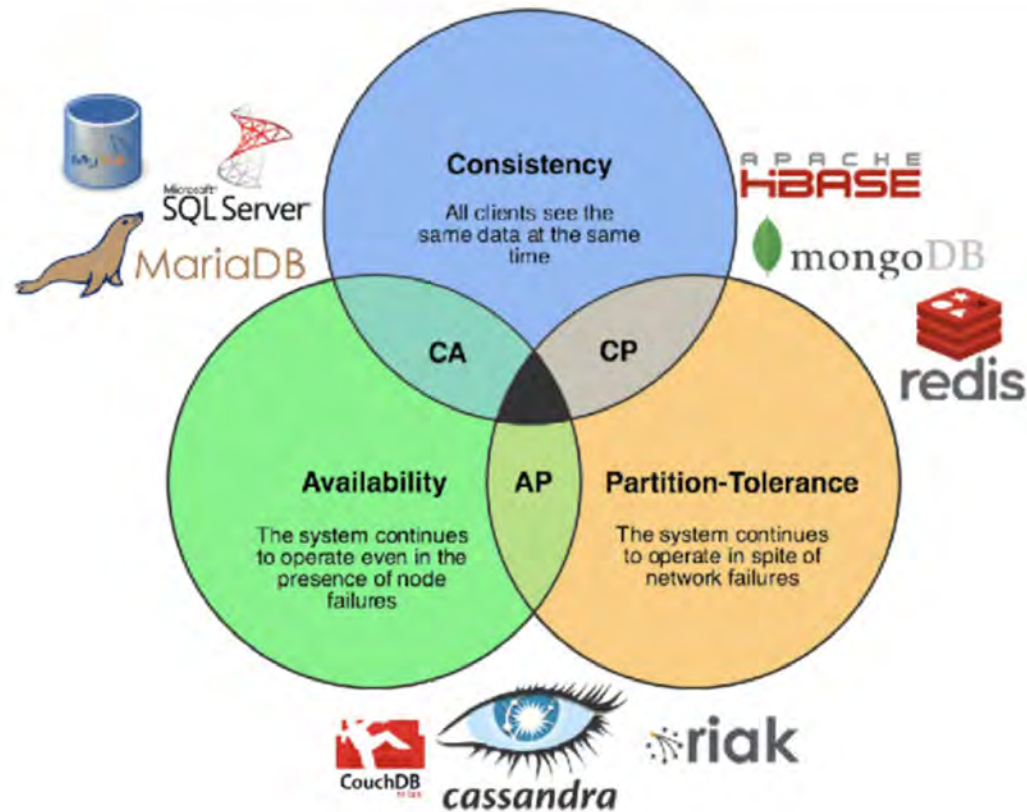
EMPLOYEE(Ssn, Fname, Minit, Lname, Bdate, Salary, Address, Sex, BossSsn, DNumber)

DEPARTMENT(DNumber, DName, NumofEmpl, ManagerSsn, Start_date)

DEP_LOC(DNumber, DLocation)

PROJECT(PNumber, Pname, PLocation, CDeptNum)

WORKSON(Ssn, PNumber, Hours)



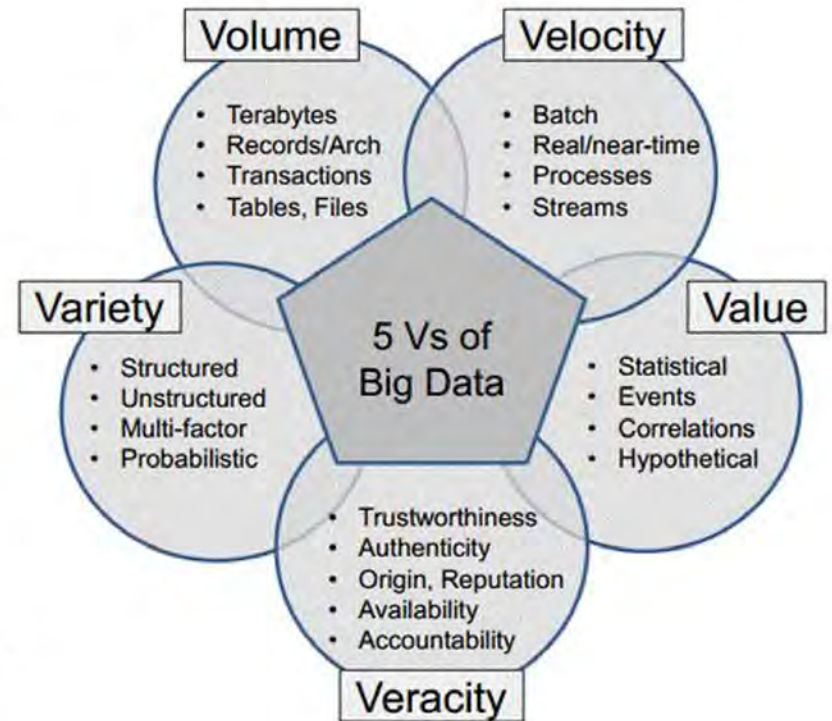
NoSQL Database Systems

Table of Content

- Handling Big Data
- CAP theorem
- NoSQL
- Types of NoSQL systems
- MapReduce framework

Big Data

- 2,5 exabyte (or more?) / day
- Big Data – **5V**:
 - **Volume**: the vast amounts of data generated every second
 - **Velocity**: speed at which new data is generated and the speed at which data moves around
 - **Variety**: variety of data types
 - **Veracity**: messiness or trustworthiness of the data
 - **Value**: big data can be turned into value



Database Scaling

- **Scalability:** refers to the capability of a system to handle a growing amount of work, or its potential to perform more total work in the same elapsed time
 - increasing system resources also increases the system's performance proportionately
- **Vertical scaling:**
 - extending the system's resources (e.g. processor)
 - adding more physical or virtual resources to the server
 - hardware (for server) costs can also be high
- **Horizontal scaling:**
 - adding more instances/nodes to the database/cluster
 - the hardware used tends to be smaller, cheaper servers
 - increases the complexity of the system
 - management of the infrastructure becomes more complex
 - many NoSQL systems support

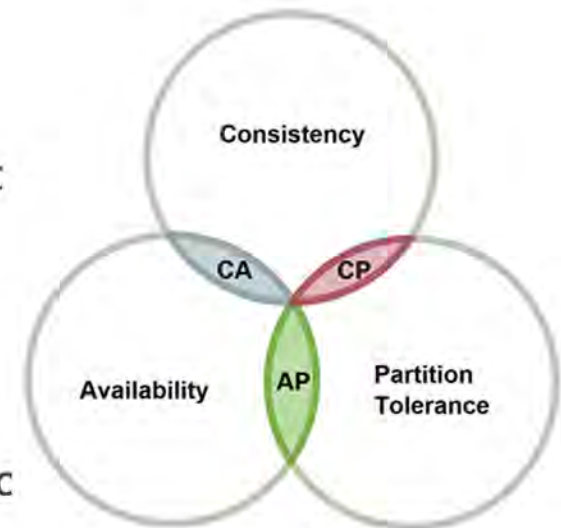
Consistency – Availability – Partition Tolerance

Requirements for distributed systems:

- **Consistency:**
 - Refers to *every client* having the *same view* of the data in the same time.
- **Availability:**
 - *Every non-failing node returns a response* for all read and write requests in a reasonable amount of time.
- **Partition tolerance:**
 - The system continues to function and upholds its consistency guarantees in spite of network partitions. Distributed systems guaranteeing partition tolerance can recover from partitions once the partition heals.

CAP Theorem

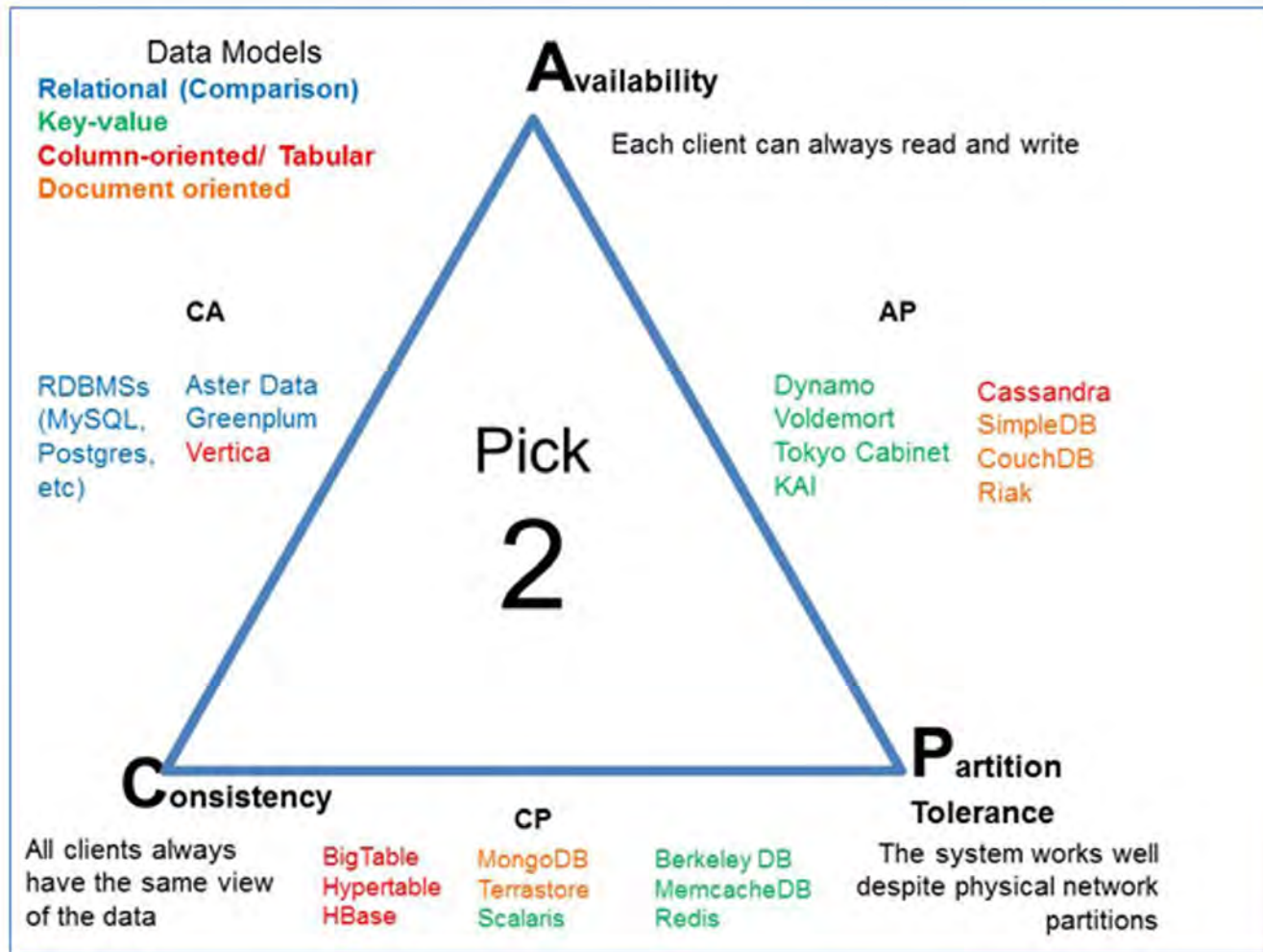
- **CAP Theorem:** In a distributed system, all three CAP properties cannot be guaranteed at the same time.
 - Consistency + Partition Tolerance
 - the system is consistent and partition tolerant but sometimes not available
 - Consistency + Availability
 - CA systems are consistent and available systems in the absence of any network partition
 - e.g.: single node DB servers
 - Availability + Partition Tolerance
 - systems that are available and partition tolerant but cannot guarantee consistency



conjecture: Eric Brewer, 2000

Theorem: Nancy Lynch, Seth Gilbert, 2002

CAP Theorem – Systems



NoSQL

- **NoSQL = Not Only SQL**

- class of **non-relational database systems**
 - Usually do not require a fixed table schema nor do they use the concept of joins.
 - e.g. BigTable (Google), Dynamo (Amazon), Cassandra (Facebook/Apache), Pnuts/Sherpa (Yahoo), CouchDB, MongoDB, ...
- mainly **distributed database systems**
- **all NoSQL systems relax one or more of the ACID properties** (Atomicity, Consistency, Isolation, Durability)
- NoSQL ≠ „Don't use SQL language“

NoSQL vs. SQL

- Advantages of NoSQL systems:
 - Flexible schema
 - Distributed computing
 - Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
 - Horizontal scalability
- What does NoSQL NOT provide?
 - ACID transactions!
 - SQL
 - Integration with applications that are based on SQL
 - Joins
 - Group by

Types of NoSQL Systems

- **Wide Column Stores**
 - each storage block contains data from only one column
- **Key-value Stores**
 - data is represented as a collection of key-value pairs,
 - such that each possible key appears at most once in the collection
- **Document Stores**
 - systems that stores documents (e.g.: XML, JSON, BSON, PDF, MS Word, YAML, ...)
- **Graph Database Systems**
 - systems that stores graphs

Wide Column Stores

Wide column stores:

- Data is organized and stored in columns (instead of rows) and each column is stored separately on disk.
 - read and write is done using columns rather than rows
 - all the cells corresponding to a column are stored as a continuous disk entry thus makes the search/access faster
 - columns are logically grouped into **column families**
- Example systems: *Hbase, BigTable, Accumulo, Vertica, ...*

Row-store vs. Column-store

Row-store:

Pro:

- easy add/modify a record

Con:

- might read a lot of unnecessary data



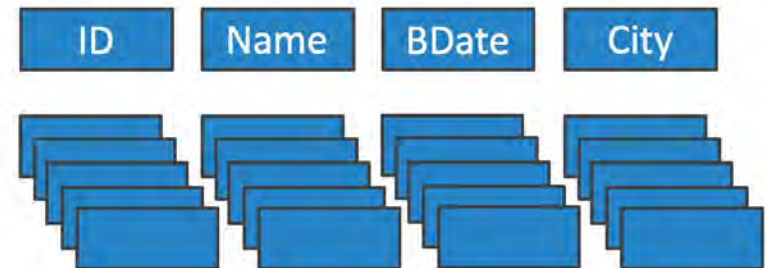
Column-store:

Pro:

- only relevant data (columns) are read

Con:

- writing a record requires multiple accesses



Wide Column Stores – Data Model

- Data model:
 - **Column:** an ordered list of elements
 - *Columns can be created at runtime* (e.g. at insertion or modification time) or during the definition of the schema.
 - **Column Family:** a single structure that groups Columns
 - It can contain a virtually unlimited number of columns.
 - *Creation of new tables and new column families is not possible during “normal operations” and has to be done only at “schema design time”.*
 - **Key:** the permanent name of the record. Keys have different numbers of columns, so the database can scale in an irregular way.
 - **Keyspace:** the outermost level of an organization, typically the name of the application (database name).

Wide Column Stores – Example Hbase Table

- Hbase Table: *User*
- Column families: *Account, Personal, Friends*
- Columns in Account: *username, password*

User			
	Account	Personal	Friends
1001	username = "bob1987" password = "thisisapassword"	firstName = "Bob" lastName = "Smith" ssn = "4hfe94"	2004:firstName = "Alice" 2004:lastName = "Smith" 2004:email = "alice@gmail.com" 1714:firstName = "Charlie" ...
2004	username = "alice"	...	...
1714	...	...	...

Key-value Stores

Key-value stores:

- designed for storing data in a **schema-less** way
- data model: **(key, value) pairs**
 - contains **objects** (called **values**) each identified by a unique **key**.
 - **key**: synthetic or auto-generated
 - **value**: can be different data types, eg.: strings, integers, lists, sets, JSON, BLOB etc., depending on the expressive power of the specific system
- Performance is enhanced to a great degree because of the cache mechanisms that accompany the mappings.
- Example systems: *Redis, Riak, Dynamo, Project Voldemort, Azure Table Storage (ATS), ...*

Key-value Store – Example

Key	Value
"India"	{"B-25, Sector-58, Noida, India – 201301"}
"Romania"	{"IMPS Moara Business Center, Buftea No. 1, Cluj-Napoca, 400606", City Business Center, Coriolan Brediceanu No. 10, Building B, Timisoara, 300011"}
"US"	{"3975 Fair Ridge Drive, Suite 200 South, Fairfax, VA 22033"}

Key-value Stores

Pros:

- records are distributed to nodes based on key → find the record for a given key is very quick
- record may be replicated across multiple nodes → efficiency, fault-tolerance

Cons:

- as the volume of data increases, maintaining unique values as keys may become more and more difficult
- not an ideal method if you are only looking to just update part of a value or query the database
- the model will not provide any kind of traditional database capabilities (such as ACID properties of transaction)

Document Stores

Document Stores:

- Like Key-value Stores except: the value is a document
 - the only difference between the key-value stores and document stores is that "documents" (values) provide some structure and encoding of the managed data
- Data model: (key, document) pairs
 - **Document:**
 - known type of structure
 - Example: JSON, XML, other semi-structured formats
- Example systems: *CouchDB, MongoDB, SimpleDB, ...*

Document Stores – Example

Embedded attribute metadata associated with stored content, which essentially provides a way to query the data based on the contents:

```
users: [
  {
    _id: "1001",
    username: "bob1987",
    password: "thisisapassword"
    personal: {
      firstName: "Bob",
      lastName: "Smith",
      ssn: "4hfe94"
    },
    friends: [
      {
        id: "2004",
        firstName: "Alice",
        lastName: "Smith",
        email: "alice@gmail.com"
      },
      {
        id: "1714",
        firstName: "Charlie",
        ...
      },
      ...
    ]
  },
  ...
]
```


Graph Database Systems

- Data model: **nodes** and **edges**
 - Nodes may have properties (often ID)
 - Edges may have labels or roles
- Interfaces and query languages vary – no standardization
 - Queries e.g:
 - single-step
 - full recursion
 - path-expression



- Example systems: *Neo4j, HypergraphDB, AllegroGraph, InfiniteGraph, ...*

MapReduce Framework

- **MapReduce:** a programming model for processing large data sets with a *parallel, distributed algorithm on clusters*
 - good technique to work on large sets of data when it is possible to work on small pieces of that dataset in parallel
- **Hadoop:**
 - widely used open source system that implements the MapReduce framework
 - distributed file system (Hadoop Distributed File System - HDFS)
 - + API for handling data
 - Used in: Yahoo, Facebook, Twitter, LinkedIn, Ebay, Microsoft, Apple,...

MapReduce Framework

MapReduce Framework:

- data processing on **distributed** servers,
- running the various tasks in **parallel**,
- managing all communications and data transfers between the various parts of the system
 - **master node**
 - **worker node**
- providing **fault tolerance** and **scalability**

- no data model, data is stored in:
 - files
 - GFS: Google File System
 - HDFS: Hadoop Distributed File System
 - (or in a database)

Parallel Computation

Parallelization: **master** and **worker** nodes

- The **master node** takes the input, chops it up into smaller sub-problems, and distributes those to **worker nodes**.
 - A worker node may do this again in turn, leading to a multi-level tree structure.
 - A master node orchestrates that for redundant copies of input data, only one is processed.
- The **worker node** processes that smaller problem, and passes the answer back to its master node.

The MapReduce Process – 3 Steps

MapReduce as a 3-step parallel and distributed computation:

- **Map:** Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage.
- **Shuffle the Mapper output to Reducers:** Worker nodes redistribute data based on the output keys (produced by the "map()" function) such that all data belonging to one key is located on the same worker node.
- **Reduce:** Worker nodes process each group of output data, per key, in parallel.

Map() and Reduce() Functions

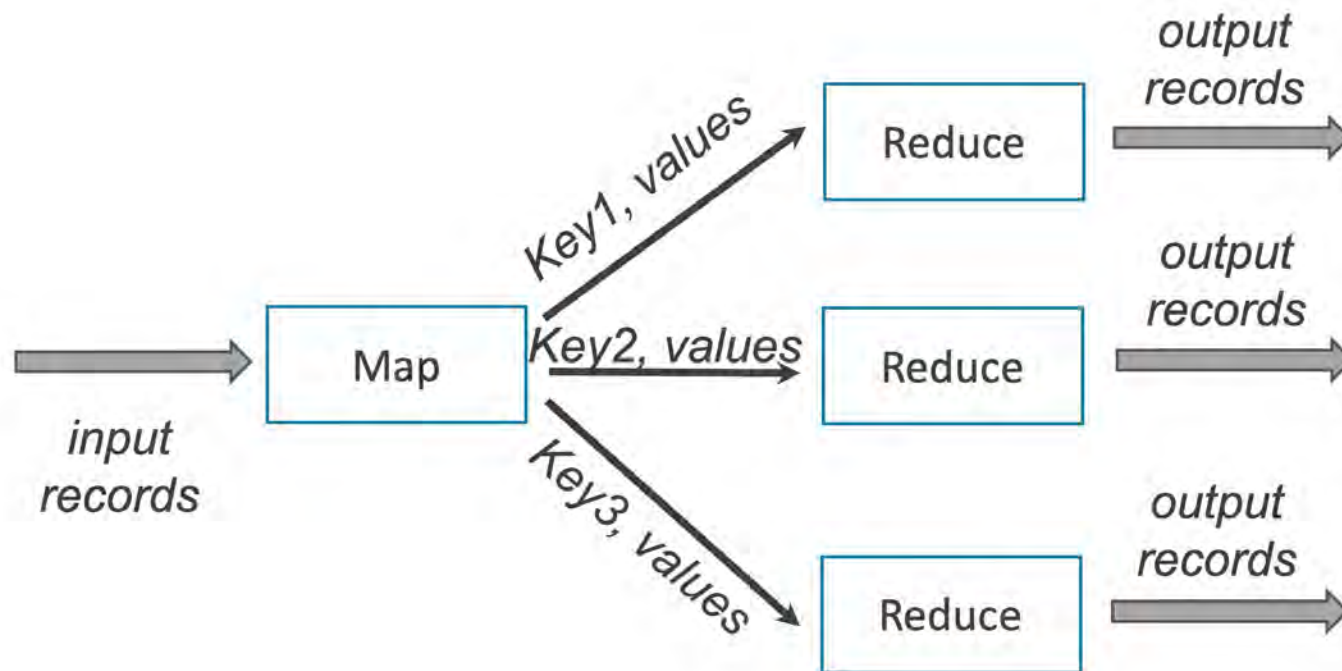
- **map():** it divides the data analysis problem into subproblems
 - the output is in the form of *key-value pairs*
 - the output can have multiple entries with the same key

map(item) → list of < key, value > pairs

- **reduce():** works on subproblems and combines the result
 - It gets values for a specific key, and generates a new list of the *reduced* output.

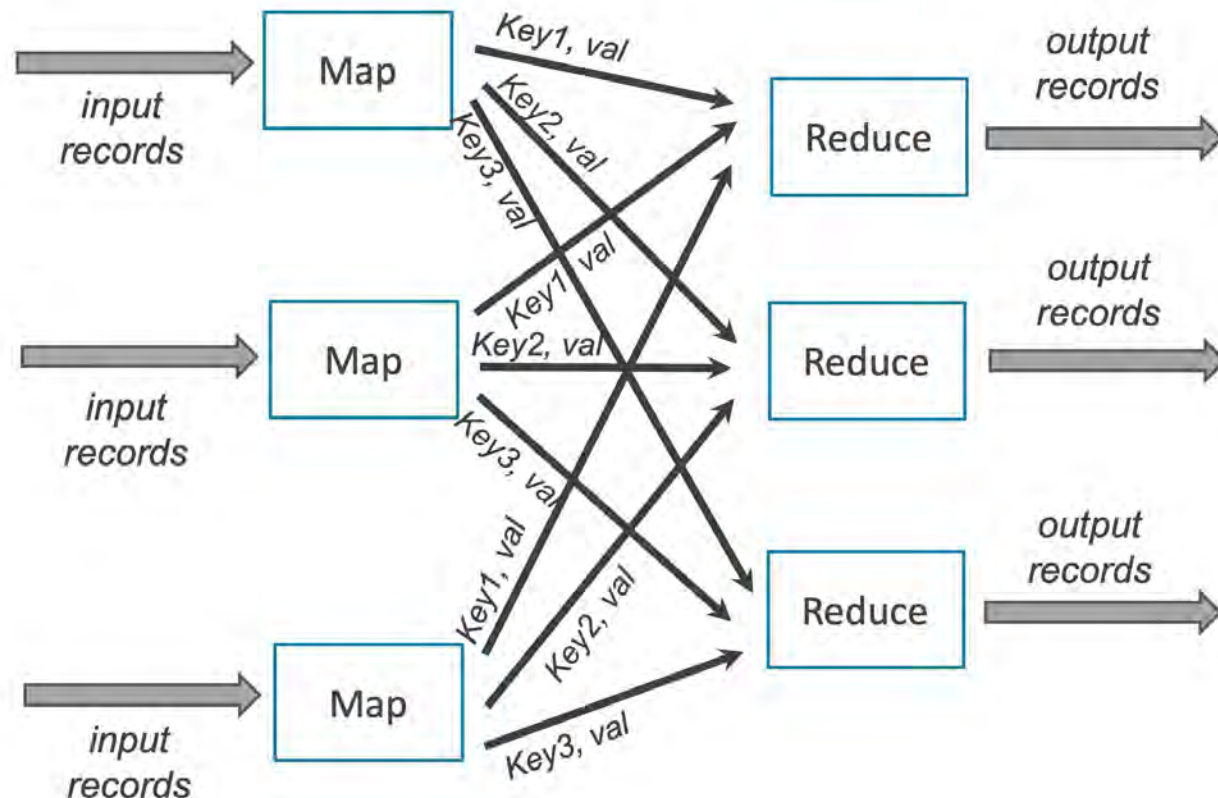
reduce(key, list of values) → 0 or more records

3-step MapReduce Architecture I.



3-step MapReduce Architecture II.

Map can be parallelized



The MapReduce Process – 5 Steps

MapReduce as a 5-step parallel and distributed computation:

- **Prepare the input for Mapper:** the MapReduce framework designates Map processors, assigns the input key(k1) value each Mapper would work on, and provides that Mapper with all input data associated with that key value.
- **Map:** Map() is run exactly once for each key (k1) value, generating output organized by key (k2) values.
- **Shuffle the Mapper output to Reducers:** assigns the key (k2) value each Reducer would work on, and provides that Reducer with all the Mapper-generated data associated with that key value.
- **Reduce:** Reduce() is run exactly once for each key (k2) value, generating the output value.
- **Produce the final output:** the MapReduce framework collects all the Reducer outputs, and sorts them by key values to produce the final outcome.

Map() and Reduce() functions (cont.)

map():

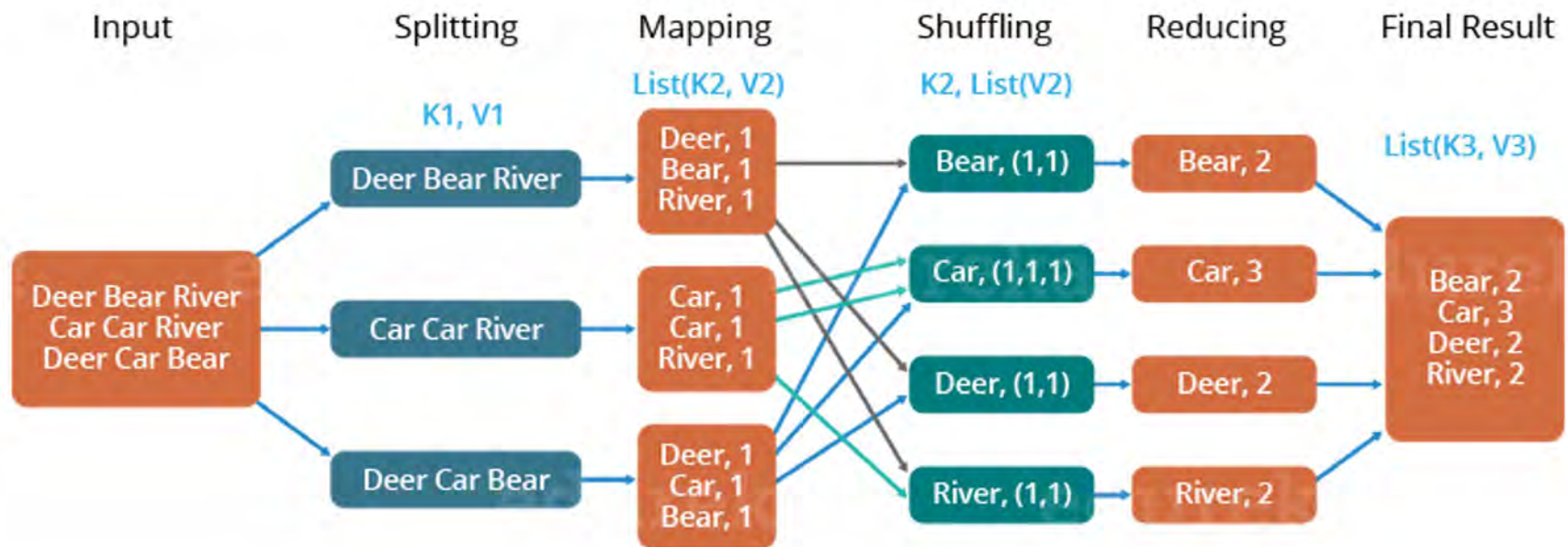
map(k1, v1) → list of < k2, v2 >

reduce():

reduce(k2, list(v2)) → list(k3, v3)

5-step MapReduce Architecture

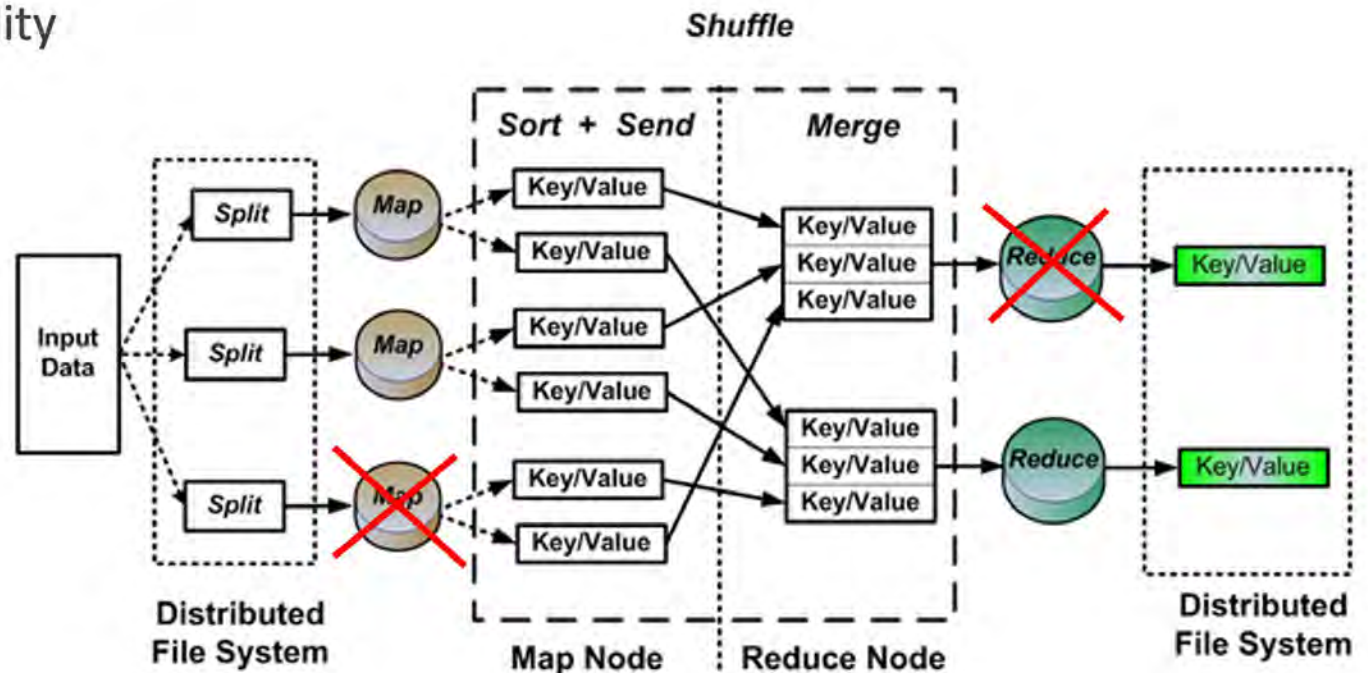
The Overall MapReduce Word Count Process



Source: <https://www.edureka.co/blog/mapreduce-tutorial/>

MapReduce Architecture

- MapReduce systems provides:
 - data processing "glue"
read/split input → map → shuffle → reduce → write output
 - fault-tolerance
 - scalability



ACID – NoSQL, NewSQL

- NoSQL databases: give up ACID (and SQL) for performance
- **NewSQL systems:**
 - a class of *modern relational database management systems* that
 - seek to provide the same *scalable performance* of NoSQL systems *for OLTP* read-write workloads
 - while still *maintaining the ACID guarantees* of a traditional database system
 - Examples: *Google Spanner, VoltDB, Clustrix, FoundationDB, NuoDB, ...*

RDBMS – NoSQL - NewSQL

Characteristic	RDBMS	NoSQL	NewSQL
ACID compliance (Data, Transaction integrity)	Yes	No	Yes
OLAP/OLTP	Yes	No	Yes
Data analysis (aggregate, transform, etc.)	Yes	No	Yes
Schema rigidity (Strict mapping of model)	Yes	No	Maybe
Data format flexibility	No	Yes	Maybe
Distributed computing	Yes	Yes	Yes
Scale up (vertical)/Scale out (horizontal)	Yes	Yes	Yes
Performance with growing data	Fast	Fast	Very Fast
Performance overhead	Huge	Moderate	Minimal
Popularity/comm unity Support	Huge	Growing	Slowly growing



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének
együttes javítása a Pannon Egyetemen

THANK YOU FOR YOUR KIND ATTENTION!

SZÉCHENYI 2020



MAGYARORSZÁG
KORMÁNYA

Európai Unió
Európai Szociális
Alap



BEFEKTETÉS A JÖVŐBE



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének
együttes javítása a Pannon Egyetemen

MS SQL 2019+ PRACTICE

Slides was compiled by
Ágnes Vathy-Fogarassy, Károly Machalik

SZÉCHENYI 2020



MAGYARORSZÁG
KORMÁNYA

Európai Unió
Európai Szociális
Alap



BEFEKTETÉS A JÖVŐBE

Introduction

MS SQL Server main services

- Relational data storage
- Data-driven application development support
- High Performance Data Integration (ETL)



- Fast and scalable multidimensional data analysis (OLAP)
- Data Mining
- Reporting

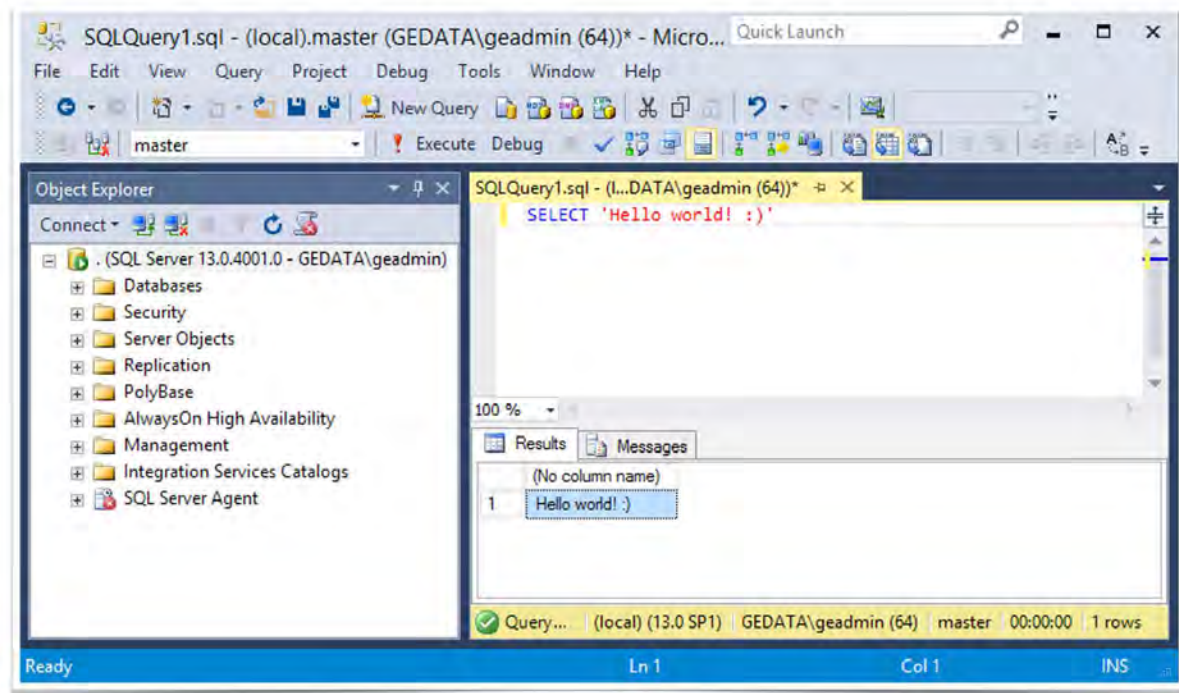


- R (2016+), Python (2017+), PolyBase (2017+), JVM (2019+)
- Windows, Linux (2017+), Azure



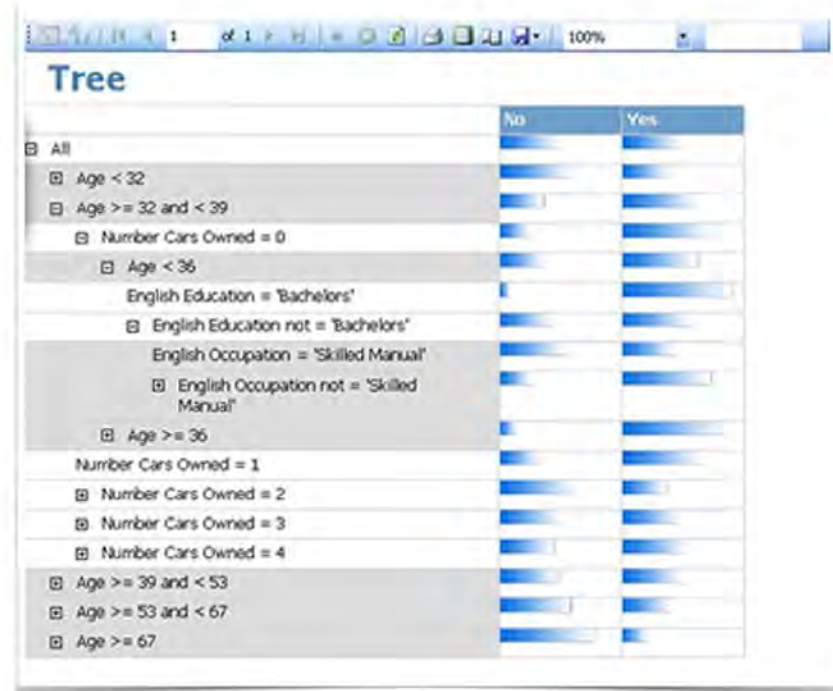
MS SQL Server main services I.

- Database Engine
 - Providing the main functionalities of relational databases
 - Server-side programming
 - Manage transactions
 - Data storage
 - ACL
 - Indexing
 - OLTP



MS SQL Server main services II.

- Analysis Services
 - Creating analytical solutions
 - Decision support
 - Data warehouse and data mining based data analysis services
 - OLAP support



MS SQL Server main services III.

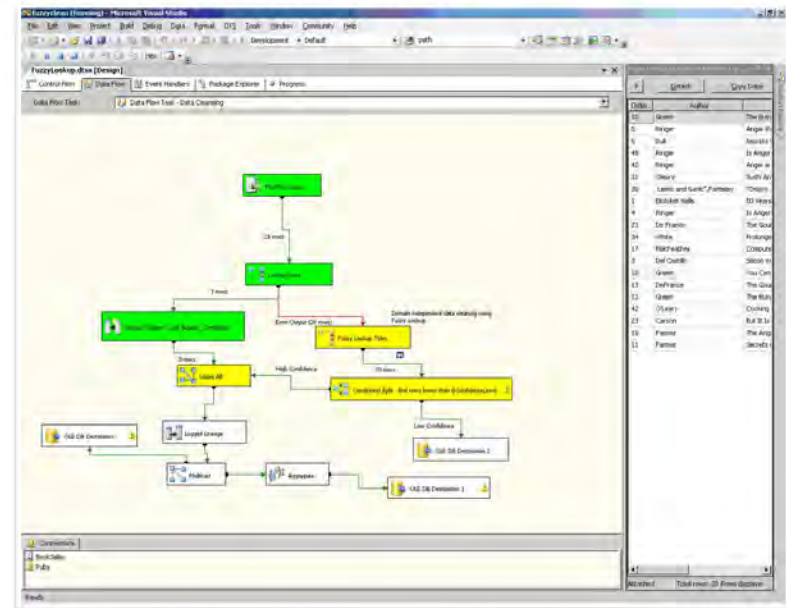
- Reporting Services
 - Interactive tabular, graphical and free form reports
 - Relational, multidimensional, XML or JSON based data sources
 - Export reports to external applications



MS SQL Server main services IV.

- Integration Services

- Data integration platform
- Unification of data from different sources (as if they were in a database - virtual database)
- Migration of databases
- Execution of data extraction, transformation and loading (ETL) processes
- Main field of application: business process automatization



MS SQL Server main services V.

- Replication
 - Support for distributed databases
 - Data sharing, synchronization, maintaining consistency, etc.
- Full-text Search
 - Support for searches on textual (unstructured) data
 - Search via SQL statements
 - Supports eg. proximity, or multiple keyword searches
 - Does not yet support Hungarian

Database files

- Databases are stored in files on the storage:
 - Primary data file (default extension is mdf):
 - Each database has one primary data file
 - Data storage
 - Secondary data files (default extension is ndf):
 - Their existence is optional
 - Data storage
 - Log file (transaction log) (default extension is ldf):
 - Storing transaction log information
- You can put information in secondary file groups, such as those that do not change (read-only), do not need to be constantly saved or modified
- Logical and physical filenames may have different mappings in the **master** database, located in the **sysdatabases** table
- Using multiple storage units can greatly improve speed

Databases

- User databases
- System databases
 - **master**: system-level settings
 - **msdb**: backups, job details, alarm scheduling
 - **model**: a model (sample) of new databases is essential for creating new databases
 - **tempdb**: temporary data storage is built up when SQL Server is started, its contents are deleted when it is shut down
 - **resource**: hidden, also used to store system objects

Some System Tables (System Catalog)

Table	Description
sys.check_constraints	Contains a row for each object that is a CHECK constraint.
sys.columns	Returns a row for each column of an object that has columns, such as views or tables.
sys.events	Contains a row for each event for which a trigger or event notification fires.
sys.foreign_keys	Contains a row per object that is a FOREIGN KEY constraint.
sys.foreign_key_columns	Contains a row for each column, or set of columns, that comprise a foreign key.
sys.identity_columns	Contains a row for each column that is an identity column.
sys.indexes	Contains a row per index or heap of a tabular object, such as a table, view, or table-valued function.
sys.index_columns	Contains one row per column that is part of a sys.indexes index or unordered table (heap).
sys.key_constraints	Contains a row for each object that is a primary key or unique constraint. Includes sys.objects.type PK and UQ.
sys.objects	Contains a row for each user-defined, schema-scoped object that is created within a database.
sys.partitions	Contains a row for each partition of all the tables and most types of indexes in the database.
sys.procedures	Contains a row for each object that is a procedure of some kind.
sys.synonyms	Contains a row for each synonym object.
sys.tables	Returns a row for each user table in SQL Server.
sys.triggers	Contains a row for each object that is a trigger, with a type of TR or TA.
sys.views	Contains a row for each view object.

Some System Tables (System Catalog)

- Database and files catalog

Table	Description
sys.databases	Contains one row per database in the instance of SQL Server.
sys.database_files	Contains a row per file of a database as stored in the database itself.
sys.filegroups	Contains a row for each data space that is a filegroup.
sys.master_files	Contains a row per file of a database as stored in the master database.

- Security and authorization

Table	Description
sys.database_permissions	Returns a row for every permission or column-exception permission in the database.
sys.database_principals	Returns a row for each security principal in a SQL Server database.
sys.server_permissions	Returns one row for each server-level permission.
sys.server_principals	Contains a row for every server-level principal.
sys.sql_logins	Returns one row for every SQL Server authentication login.

Some System Tables (System Catalog)

- Dynamic management

Table	Description
sys.dm_db_index_usage_stats	Returns counts of different types of index operations and the time each type of operation was last performed.
sys.dm_exec_cached_plans	Returns a row for each query plan that is cached by SQL Server for faster query execution.
sys.dm_exec_query_stats	Returns aggregate performance statistics for cached query plans in SQL Server.

- Other

Table	Description
sys.messages	Contains a row for each message_id or language_id of the error messages in the system, for both system-defined and user-defined messages.
sys.server_events	Contains one row for each event for which a server-level event-notification or server-level DDL trigger fires.
sys.server_event_sessions	Lists all the event session definitions that exist in SQL Server.
sys.servers	Contains a row per linked or remote server registered.

Main objects of databases

- **Tables**

- Organized into partitions - usually 1, but for large data tables it is advisable to partition them
 - Tables can be defined as:
 - data types
 - keys and restrictions
 - default values

- **Views:** virtual tables, does not exist physically

- **Indices**

- **Synonyms:** reference names typically to objects in an external schemas

- **Stored Procedures and Functions:** A collection of T-SQL statements stored on a server-side

- **Triggers:** Stored procedures triggered by an event which cannot be called directly

Users, Roles, Permissions

- **There are two methods of authentication**
 - Windows authentication (on Windows OS only)
 - SQL Server authentication
 - sa login: system administrator
- **Users**
 - User authentication: username + password
 - Default database can be assigned to each user
- **Roles**
 - Users are organized into roles
- **Rights**
 - User and role level privileges

Roles

- **Role**
 - Authorization group to which users and permissions can be assigned.
- Roles may be defined on different levels
 - ***server*** level
 - ***database*** level
 - ***application roles*** new applications can be defined based on the needs of the applications

Some important built-in roles

Level	Role	Description
server	sysadmin	Members of the sysadmin fixed server role can perform any activity in the server.
server	securityadmin	Members of the securityadmin fixed server role manage logins and their properties. They can GRANT, DENY, and REVOKE server-level permissions. They can also GRANT, DENY, and REVOKE database-level permissions if they have access to a database. Additionally, they can reset passwords for SQL Server logins.
server	dbcreator	Members of the dbcreator fixed server role can create, alter, drop, and restore any database.
server	public	Every SQL Server login belongs to the public server role.
db	db_owner	Members of the db_owner fixed database role can perform all configuration and maintenance activities on the database, and can also drop the database.
db	db_securityadmin	Members of the db_securityadmin fixed database role can modify role membership and manage permissions. Adding principals to this role could enable unintended privilege escalation.
db	db_datawriter	Members of the db_datawriter fixed database role can add, delete, or change data in all user tables.
db	db_denydatawriter	Members of the db_denydatawriter fixed database role cannot add, modify, or delete any data in the user tables within a database.

Rights

- Example: Main permissions associated with a table:

Permissions for guest: Column Permissions...

☐ Explicit ☒ Effective

Permission	Grantor	Grant	With Grant	Deny
Alter		☐	☐	☐
Control		☐	☐	☐
Delete		☐	☐	☐
Insert		☐	☐	☐
References		☐	☐	☐
Select		☐	☐	☐
Take ownership		☐	☐	☐
Update		☐	☐	☐
View change tracking		☐	☐	☐
View definition		☐	☐	☐

- The permissions
 - Grant
 - Grant with forwarding privileges
 - Deny

Schemes, Namespaces and Users

- Starting with version 2005, the schema and users are separated.
- ***Schema*** = object container
 - Database objects exist in schemas.
- Removing a user does not require renaming the schema.
- A database can contain multiple schema!
- Each schema has an owner - user or role.
- All users have a default schema that they may not own.
- The ownership chain is still operating.

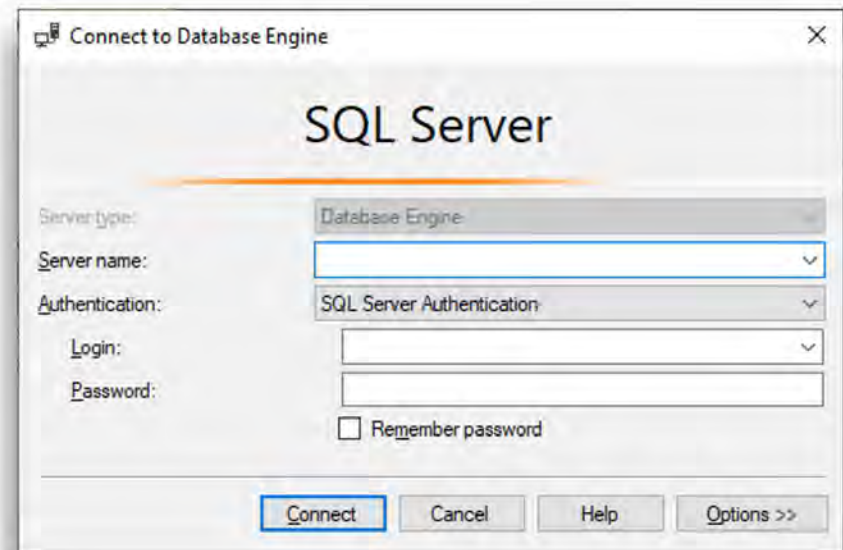
Databases, Datatypes

SQL syntax

- The SQL language is not case sensitive
- The semicolon (;) closes the statement
- The statements can be wrapped arbitrarily
- **Comment:**
 - One line: --
 - Block: / * ... * /
- **Literal**
 - Numeric literals cannot use delimiters!
 - Each non-numerical literal must be enclosed between 2 apostrophes (e.g. 'Veszprem')

Connection & Login

- **SQL Server Management Studio (Windows)**
- Azure Data Studio, DataGrip (Multiplatform)
- **Connecting to the server**
 - Server name: IP address or DNS name of the server
 - Authentication: SQL Server Authentication
 - Login: username
 - Password: user's password



Connection & Login

Connection Details

Connection type: Microsoft SQL Server

Server:

Authentication type: SQL Login

User name:

Password:

☒ Remember password

Database: <Default>



Data Sources and Drivers

Name: Adatb1 (2019) [Reset](#)

Comment:

General Options SSH/SSL Schemas Advanced

Connection type: default Driver: Microsoft SQL Server

Host: Port:

Instance:

User:

Password: <hidden> Save: Forever [⬇](#)

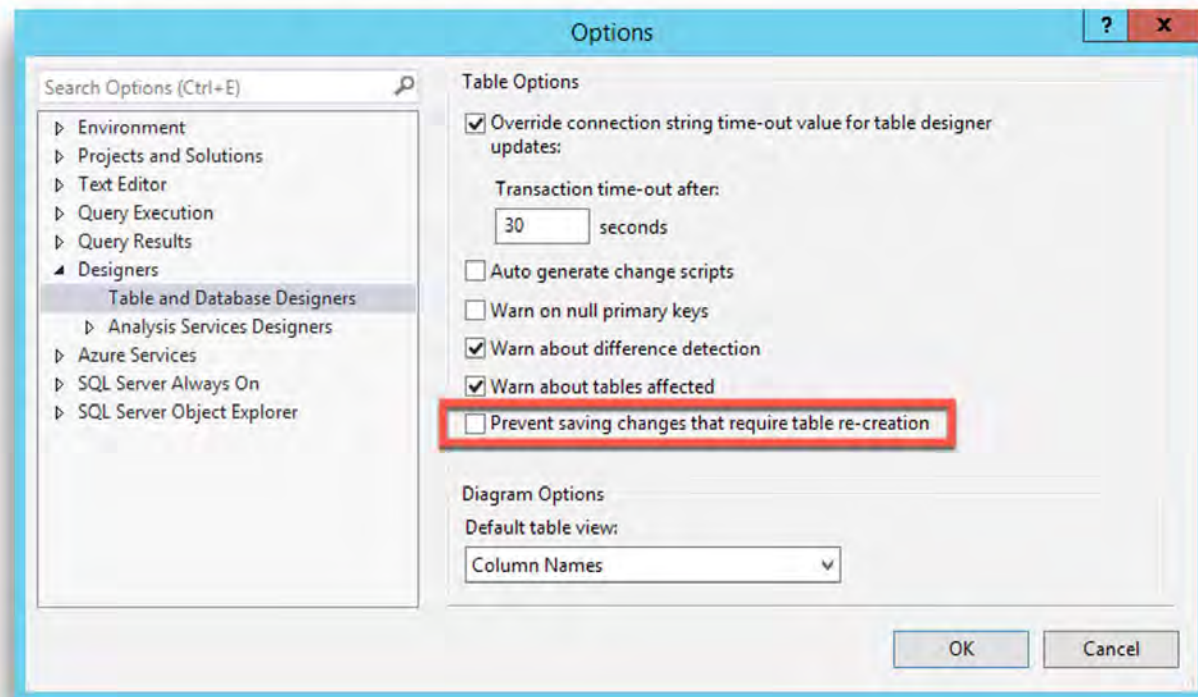
Database:

URL: jdbc:sqlserver:
Overrides settings above.

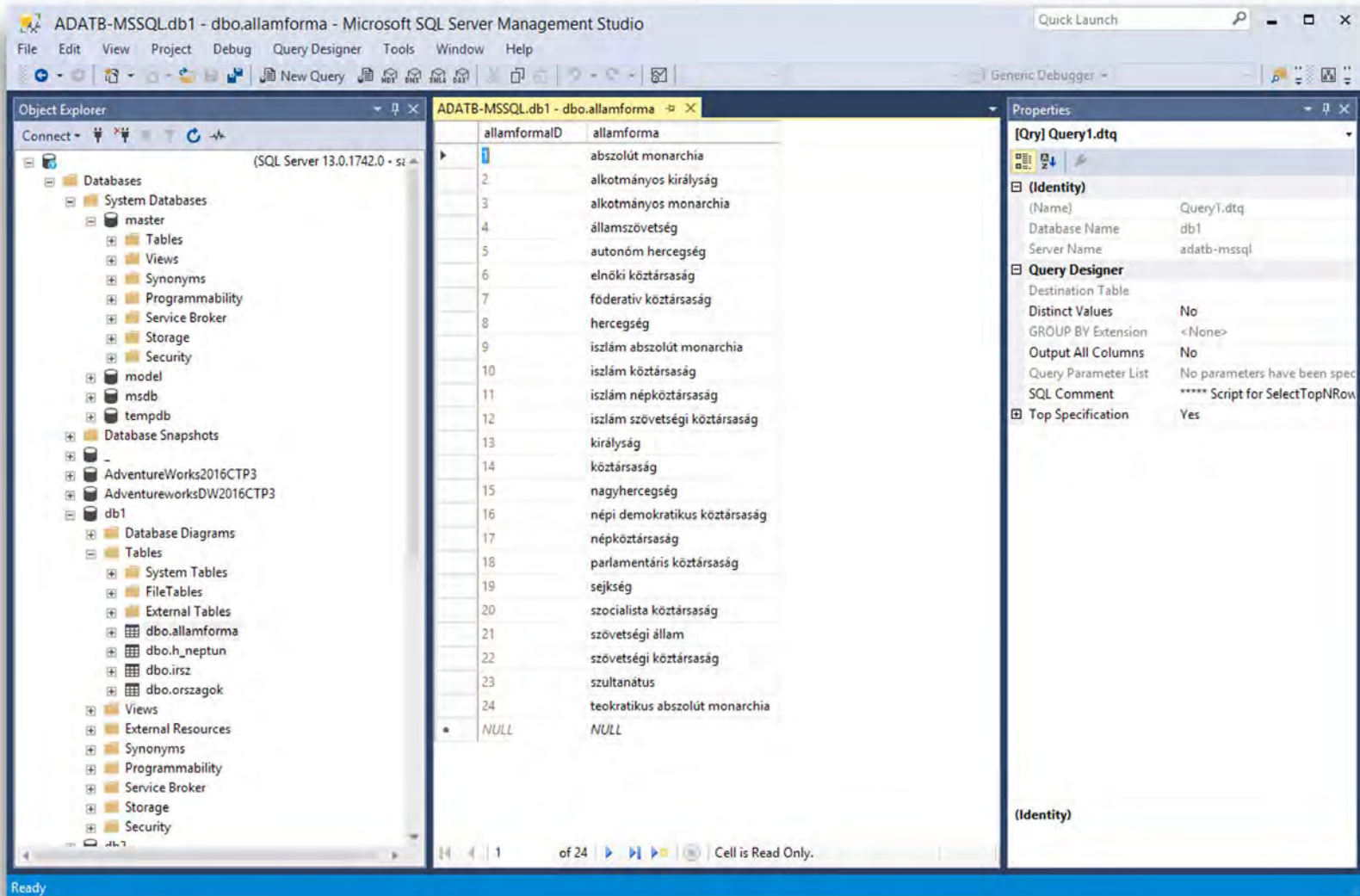
[Test Connection](#)

Recommended settings

- **Management Studio**
 - **Tools / Options**
 - **Designers / Table and Database Designers**
 - **Prevent saving changes that require table re-creation**

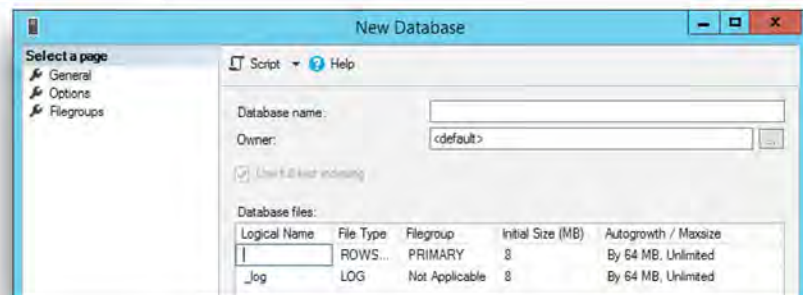
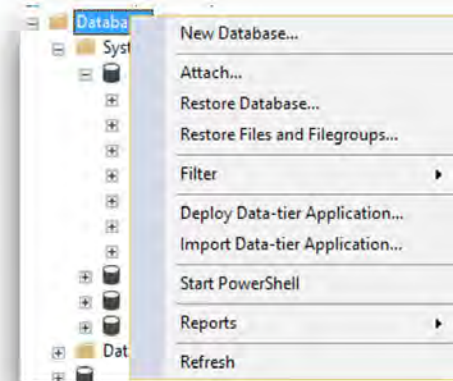


Management Studio - GUI



Create a new database - GUI

Required parameters	
Database Name	Name of the new Database
Owner	Database Owner (Optional)
Logical Name	The logical name of the database is created automatically and can be modified
File Type	Type of the Data (Log / Database)
Filegroup	Filegroup. There is no when create a filegroup.
Initial Size	Initial size of the Database.
Autogrowth/Maxsize	Way to grow your database - When the file is full, how to increase the size of SQL Server? - What percentage? Fixed size?
Path	Path to files
File name	The name of the files
Options	Database options
Filegroups	File group settings

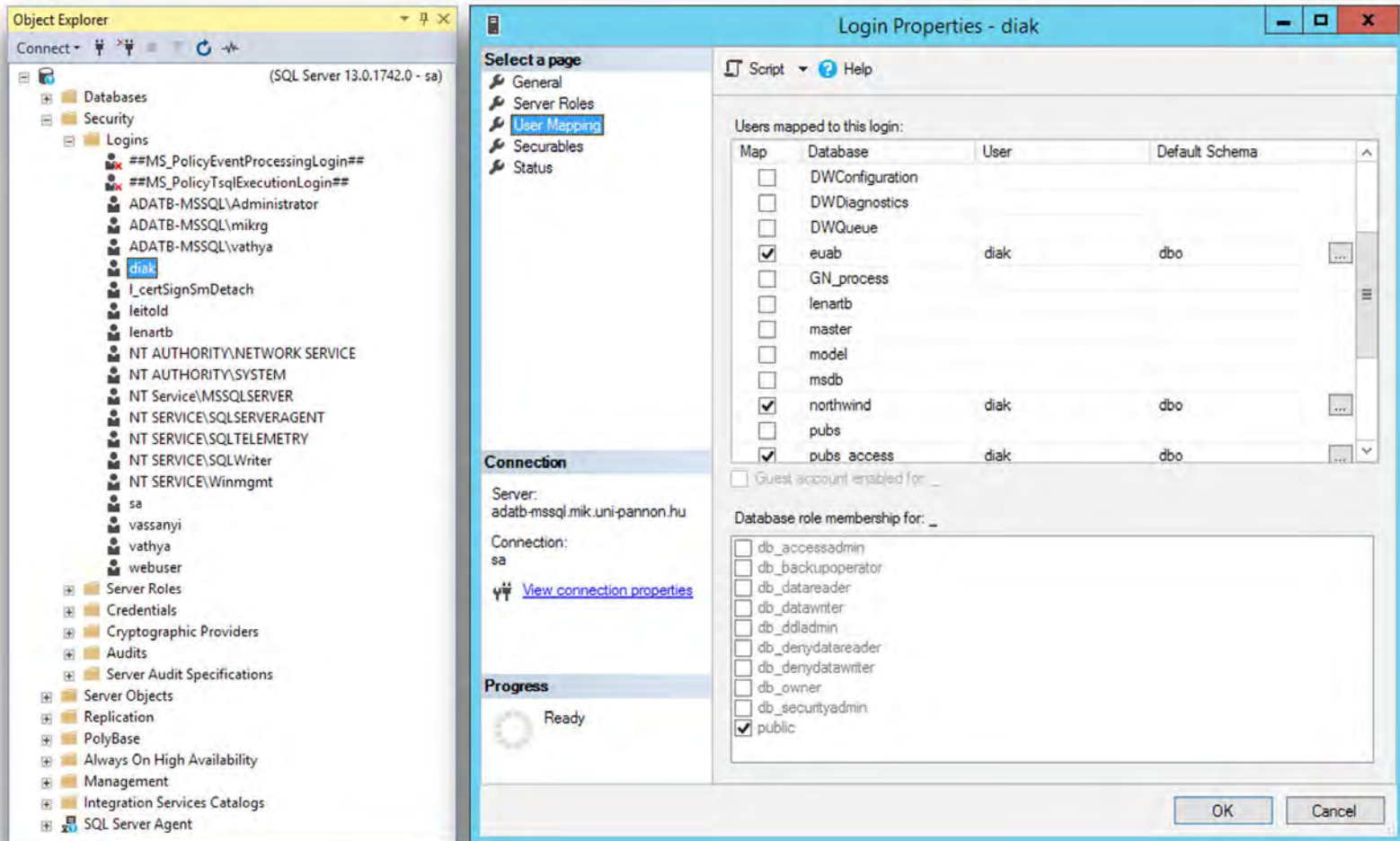


Create a new database - TSQL

```
CREATE DATABASE [db1] ON PRIMARY (  
    NAME = N'db1',  
    FILENAME = N'C:\DATA\db1.mdf',  
    SIZE = 73728 KB,  
    MAXSIZE = UNLIMITED,  
    FILEGROWTH = 65536 KB)  
LOG ON (  
    NAME = N'db1_log',  
    FILENAME = N'C:\DATA\db1_log.ldf',  
    SIZE = 8192 KB,  
    MAXSIZE = 2048 GB,  
    FILEGROWTH = 65536 KB)
```

N prefix: converts the following text constants to nvarchar (2 bytes Unicode)

Users and Permissions



Basic data types

bigint	binary(n)	bit	char(n)
date	datetime	datetime2[(n)]	datetimeoffset
decimal	float	image	int
money	nchar(n)	ntext	numeric
nvarchar(n max)	real	smalldatetime	smallint
smallmoney	sql_variant	text	time
tinyint	uniqueidentifier	varbinary(n max)	varchar(n max)

Data Types - String I.

Data type	Explain
char[(n)]	Fixed-length, non-Unicode string data. n defines the string length and must be a value from 1 through 8,000. The storage size is n bytes. The ISO synonym for char is character.
varchar[(n max)]	Variable-length, non-Unicode string data. n defines the string length and can be a value from 1 through 8,000. max indicates that the maximum storage size is $2^{31}-1$ bytes (2 GB). The storage size is the actual length of the data entered + 2 bytes. The ISO synonyms for varchar are charvarying or charactervarying.
text	Variable-length non-Unicode data in the code page of the server and with a maximum string length of $2^{31}-1$ (2,147,483,647). When the server code page uses double-byte characters, the storage is still 2,147,483,647 bytes. Depending on the character string, the storage size may be less than 2,147,483,647 bytes.

The text data type will be removed in a future version of SQL Server.

Data Types - String II.

Data type	Explain
nchar[(n)]	Fixed-length Unicode string data. n defines the string length and must be a value from 1 through 4,000. The storage size is two times n bytes. When the collation code page uses double-byte characters, the storage size is still n bytes. Depending on the string, the storage size of n bytes can be less than the value specified for n. The ISO synonyms for nchar are national char and national character.
nvarchar[(n max)]	Variable-length Unicode string data. n defines the string length and can be a value from 1 through 4,000. max indicates that the maximum storage size is $2^{31}-1$ bytes (2 GB). The storage size, in bytes, is two times the actual length of data entered + 2 bytes. The ISO synonyms for nvarchar are national char varying and national character varying.
ntext	Variable-length Unicode data with a maximum string length of $2^{30} - 1$ (1,073,741,823) bytes. Storage size, in bytes, is two times the string length that is entered. The ISO synonym for ntext is national text.

The ntext data type will be removed in a future version of SQL Server.

Data Types - Numeric I.

Data type	Range		Storage
bigint	-2^{63} (-9,223,372,036,854,775,808)	$2^{63}-1$ (9,223,372,036,854,775,807)	8 Bytes
int (primary)	-2^{31} (-2,147,483,648)	$2^{31}-1$ (2,147,483,647)	4 Bytes
smallint	-2^{15} (-32,768)	$2^{15}-1$ (32,767)	2 Bytes
tinyint	0	255	1 Byte
money	-922,337,203,685,477.5808	922,337,203,685,477.5807	8 Bytes
smallmoney	- 214,748.3648	214,748.3647	4 Bytes

Data Types - Numeric II.

- **decimal**[(p[,s])] and **numeric**[(p[,s])]
- **Fixed precision** and scale numbers. When maximum precision is used, valid values are from $-10^{38} + 1$ through $10^{38} - 1$. The ISO synonyms for **decimal** are **dec** and **dec(p, s)**. **numeric** is functionally equivalent to **decimal**.

Precision	Storage
1 - 9	5 Bytes
10 - 19	9 Bytes
20 - 28	13 Bytes
29 - 38	17 Byte

Data Types - Numeric III.

Data type	Range					Storage
float[(n)]	- 1.79E + 308	-2.23E - 308	0	2.23E - 308	1.79E + 308	<i>Depends on the value of n</i>
real	- 3.40E + 38	-1.18E - 38	0	1.18E - 38	3.40E + 38	4 Bytes

Value of n	Precision	Storage
1 - 24	7 digits	4 Bytes
25 - 53	15 digits	8 Bytes

ISO:

real **float(24)**

double precision **float(53)**

Data Types - Date and Time

Data type	Range		Precision	Storage
date	0001-01-01	9999-12-31	One day	3 Bytes
datetime	1753-01-01 00:00:00.000	9999-12-31 23:59:59.997	Rounded to increments of. 000, .003, or .007 seconds	8 Bytes
datetime2[(prec)]	0001-01-01 00:00:00.0	9999-12-31 23:59:59.9*	100 ns (7 digits) < 3 digits 3-4 digits > 4 digits	6 Bytes 7 Bytes 8 Bytes
datetimeoffset	0001-01-01 00:00:00.0	9999-12-31 23:59:59.9*	100 ns (7 digits)	10 Bytes
smalldatetime	1900-01-01 00:00:00	2079-06-06 23:59:00	One minute	4 Bytes
time	00:00:00.0	23:59:59.9*	100 ns (7 digits)	5 Bytes

Data Types - Binary

Data type	Explain
binary[(n)]	Fixed-length binary data with a length of n bytes, where n is a value from 1 through 8,000. The storage size is n bytes.
varbinary[(n max)]	Variable-length binary data. n can be a value from 1 through 8,000. max indicates that the maximum storage size is $2^{31}-1$ bytes. The storage size is the actual length of the data entered + 2 bytes. The data that is entered can be 0 bytes in length. The ANSI SQL synonym for varbinary is binary varying.
image	Variable-length binary data from 0 through $2^{31}-1$ (2,147,483,647) bytes.

The image data type will be removed in a future version of SQL Server.

Data Types - Other

Data type	Explain
cursor	Reference to a cursor
hierarchyid	The hierarchyid data type is a variable length, system data type
sql_variant	A data type that stores values of various SQL Server-supported data types
table	Is a special data type that can be used to store a result set for processing at a later time
timestamp	timestamp is the synonym for the rowversion data type
uniqueidentifier	A unique identifier (GUID, newid())
xml	Store xml instances in a column, or a variable of xml type
Spatial Types	Geography spatial data type
json	JSON data type

Data Query Language

SSMS – Useful shortcuts

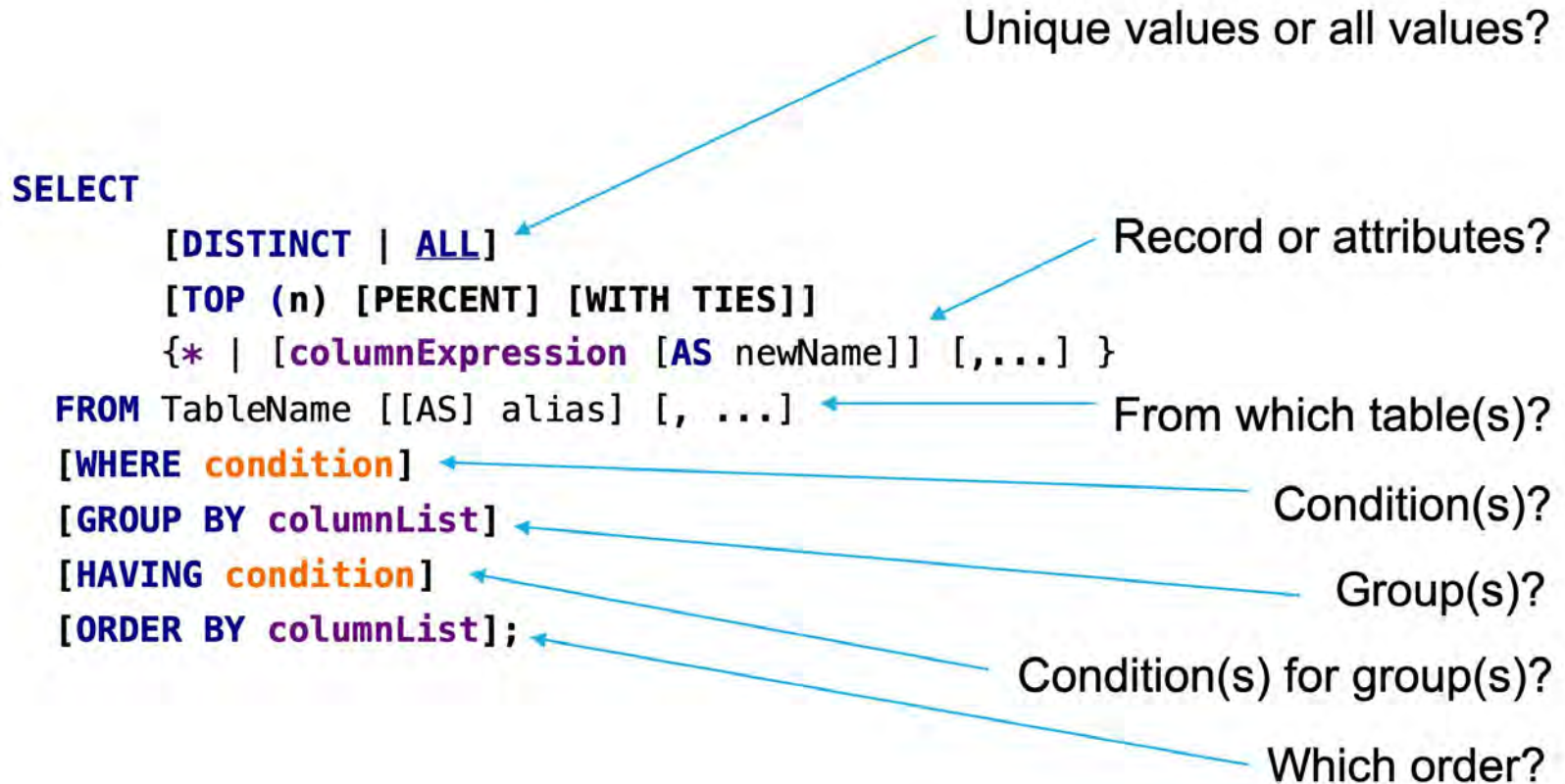
Shortcut	Function
F5	Run the SQL script
Ctrl + Shift + R	Refresh the IntelliSense cache
Ctrl + K, Ctrl + C	Comment the line or the selection
Ctrl + K, Ctrl + U	Clear comment
Ctrl + R	Show/hide the results pane
Ctrl + L	Show the Execution Plan
Ctrl + T	The result will be text
Ctrl + D	The result will be grid
Ctrl + Shift + F	Save the result to file

Query – SELECT

```
SELECT Class, AVG(Age) AS Average  
FROM Students  
WHERE Name LIKE 'A%'  
GROUP BY Class  
HAVING AVG(Age) > 7  
ORDER BY Average
```



Query – SELECT

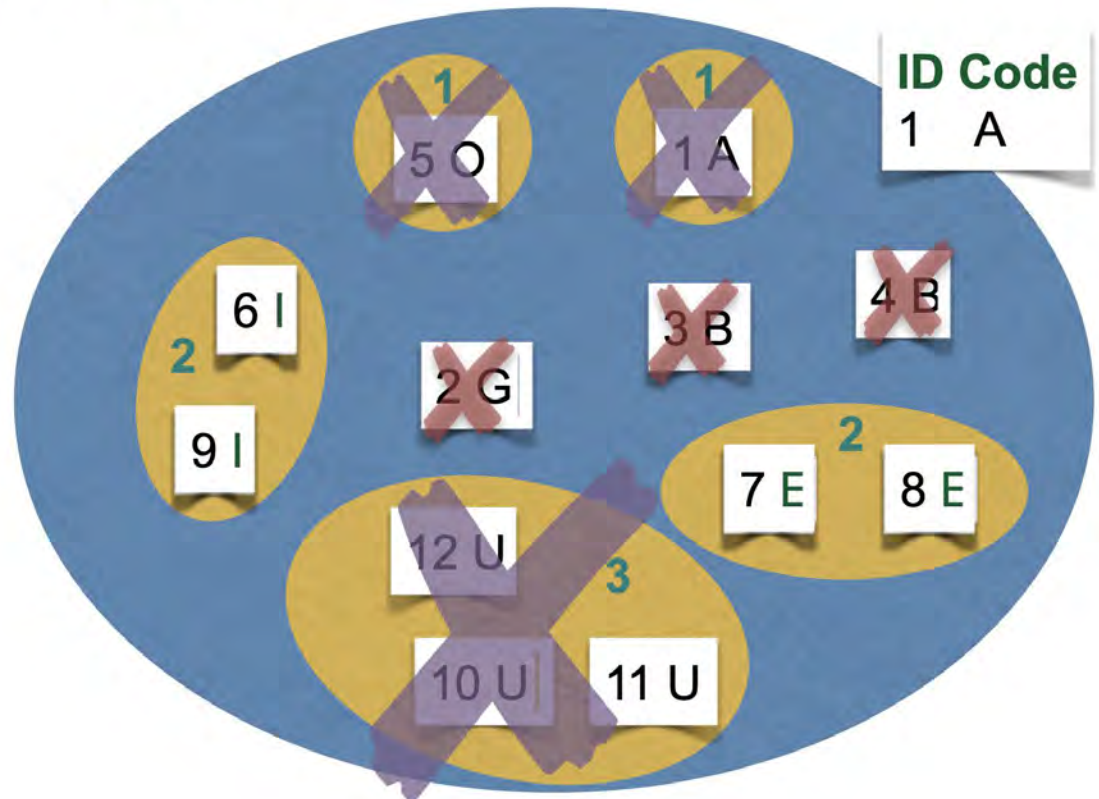


Query – SELECT

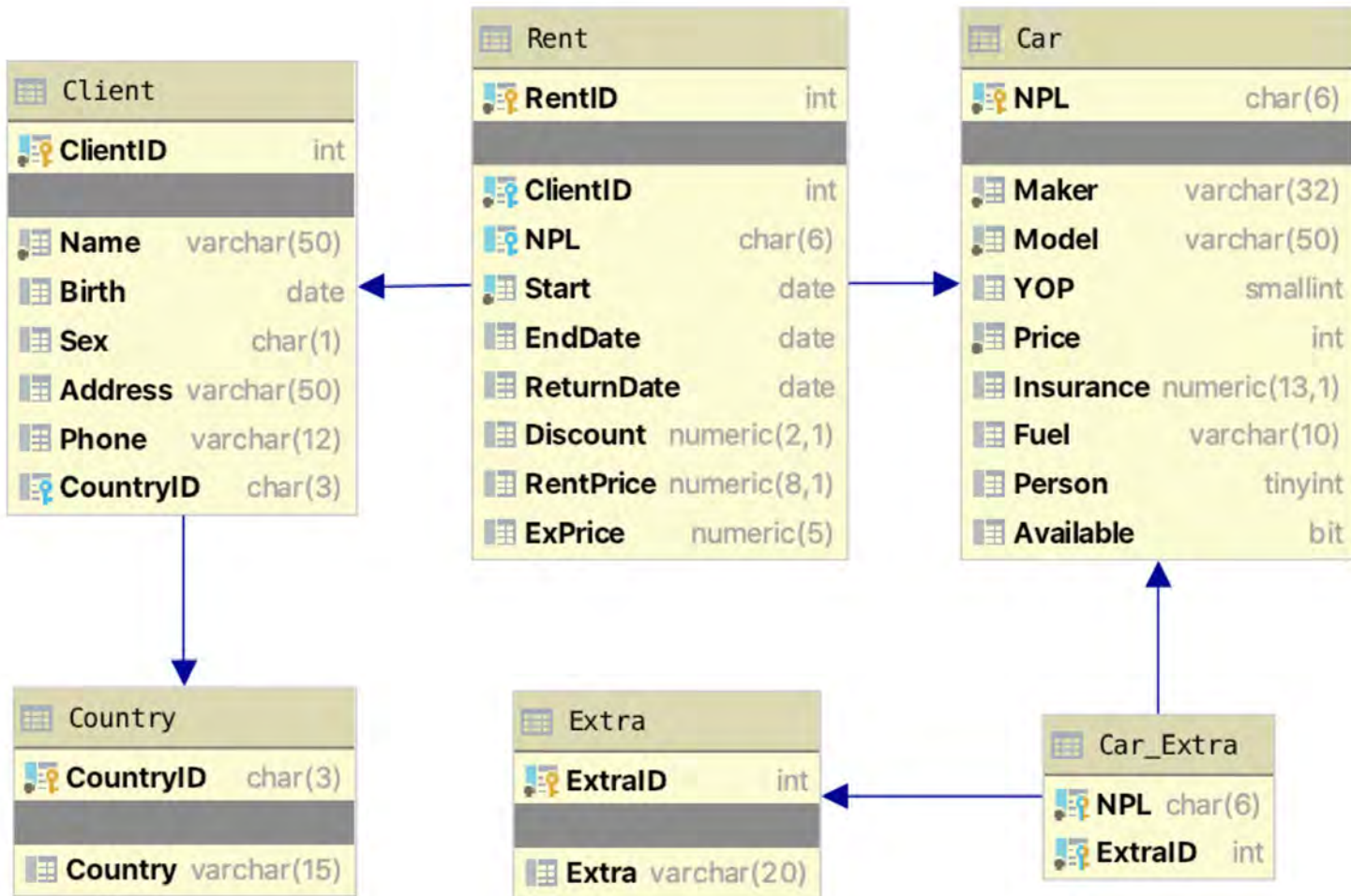
```
SELECT Code
FROM BlueSet
WHERE Code IN ('A','E','I','O','U')
GROUP BY Code
HAVING COUNT(*) = 2
ORDER BY Code
```



Code
E
I



Database schema



Query – Result set of SELECT

- Result of queries: a recordset

```
SELECT LPN, Maker, Model, YOP  
FROM car
```

LPN	Maker	Model	YOP
CBS832	BMW	Mini	2012
FKN783	Opel	Astra	2015
GHJ923	Mazda	Mazda 5	2014
GHL793	Mazda	Mazda 5	2014
GHV199	Volkswagen	Touran	2014

Query – SELECT special cases

- One record, multiple attributes

```
SELECT LPN, Maker, Model, YOP
FROM car
WHERE LPN = 'CBS832'
```

LPN	Maker	Model	YOP
CBS832	BMW	Mini	2012

- One attribute, multiple records

```
SELECT LPN
FROM car
WHERE YOP = 2014
```

LPN
GHJ923
GHL793
GHV199

- One record, one attribute

```
SELECT YOP
FROM car
WHERE LPN = 'CBS832'
```

YOP
2012

Automatic conversion: SCALAR!

Query – SELECT special cases

- Empty record

```
SELECT LPN, Maker, Model, YOP  
FROM car  
WHERE LPN = 'ABC123'
```

LPN	Maker	Model	YOP
-----	-------	-------	-----

- Empty attribute

```
SELECT LPN  
FROM car  
WHERE Maker = 'Ferrari'
```

LPN

Common functions

Function	Description	Version
[TRY_]CAST	Convert an expression of one data type to another	
GETDATE	Returns the current database system timestamp as a datetime value without time zone offset.	
YEAR	Returns an integer that represents the year of the specified date.	
MONTH	Returns an integer that represents the month of the specified date.	
DAY	Returns an integer that represents the day of the specified date.	
DATEPART	Returns an integer representing the specified datepart of the specified date.	
DATEDIFF	Returns the count (as a signed integer value) of the specified datepart boundaries crossed between the specified start and enddate.	
DATEADD	Adds a specified number value (as a signed integer) to a specified datepart of an input date value, and then returns that modified value.	
LEN	The number of characters of the specified string expression, excluding trailing spaces.	
SUBSTRING	Returns part of a character, binary, text, or image expression.	
CONCAT	Returns a string resulting from the concatenation.	2012
STRING_SPLIT	A table-valued function that splits a string into rows of substrings.	2016
CONCAT_WS	Returns a string resulting from the concatenation. It separates those concatenated string values with the delimiter specified in the first function argument.	2017
STRING_AGG	Concatenates the values of string expressions and places separator values between them.	2017
TRIM	Removes the space character char(32) or other specified characters from the start and end of a string.	2017

NULL – The “bugbear”



```
SELECT *  
  FROM null_table  
 WHERE code = NULL
```



```
SELECT *  
  FROM null_table  
 WHERE code != NULL
```



```
SELECT *  
  FROM null_table
```

▽

```
SELECT code + '@' + name AS cn, CONCAT(code, '@', name) AS cn2  
  FROM null_table  
 WHERE code IS NULL
```



▽

```
SELECT code + '@' + name AS cn, CONCAT(kod, '@', nev) AS cn2  
  FROM null_table  
 WHERE kod IS NOT NULL
```



IN - NOT IN

```
SELECT NPL, Maker, Model
FROM car
WHERE NPL IN ('CBS832', 'FKN783', 'GHL793')
```



CBS832	BMW	Mini
FKN783	Opel	Astra
GHL793	Mazda	Mazda 5

```
SELECT DISTINCT NPL
FROM rent
WHERE start BETWEEN '2015-02-01' AND '2015-08-31'
```



CBS832
FKN783
GHL793
GHV199
GJK663
GKP729
HLI190
JJJ673
KKL299
KKL556
LLK312
LRR154
LTT532
NNN724

```
SELECT NPL, Maker, Model
FROM car
WHERE NPL IN (SELECT DISTINCT NPL
               FROM rent
               WHERE start BETWEEN '2015-02-01' AND '2015-08-31')
```



CBS832	BMW	Mini
FKN783	Opel	Astra
GHL793	Mazda	Mazda 5
GHV199	Volkswagen	Touran
GJK663	Chevrolet	Spark
GKP729	Skoda	Fabia

JOIN I.

ID1	Code		ID2	Name
1	A	X	1	Alpha
2	B		4	Bravo
3	C		5	Charlie

- Cartesian product

ID1	Code	ID2	Name
1	A	1	Alpha
2	B	1	Alpha
3	C	1	Alpha
1	A	4	Bravo
2	B	4	Bravo
3	C	4	Bravo
1	A	5	Charlie
2	B	5	Charlie
3	C	5	Charlie

- CROSS JOIN

JOIN I.

ID1	Code
1	A
2	B
3	C

ID1 = ID2

ID2	Name
1	Alpha
4	Bravo
5	Charlie

- JOIN

ID1	Code	ID2	Name
1	A	1	Alpha

- LEFT [*OUTER*] JOIN

ID1	Code	ID2	Name
1	A	1	Alpha
2	B	NULL	NULL
3	C	NULL	NULL

- RIGHT [*OUTER*] JOIN

ID1	Code	ID2	Name
1	A	1	Alpha
NULL	NULL	4	Bravo
NULL	NULL	5	Charlie

- FULL [*OUTER*] JOIN

ID1	Code	ID2	Name
1	A	1	Alpha
2	B	NULL	NULL
3	C	NULL	NULL
NULL	NULL	4	Bravo
NULL	NULL	5	Charlie

JOIN III.

```
SELECT NPL, Maker, Model, price
FROM car
```

CBS832	BMW	Mini	5200
FKN783	Opel	Astra	6700
GHJ923	Mazda	Mazda 5	9800
GHL793	Mazda	Mazda 5	9800
GHV199	Volkswagen	Touran	9900
GJK663	Chevrolet	Spark	4600

```
SELECT NPL, start, rentprice, exprice
FROM rent
WHERE start BETWEEN '2015-02-01' AND '2015-08-31'
```

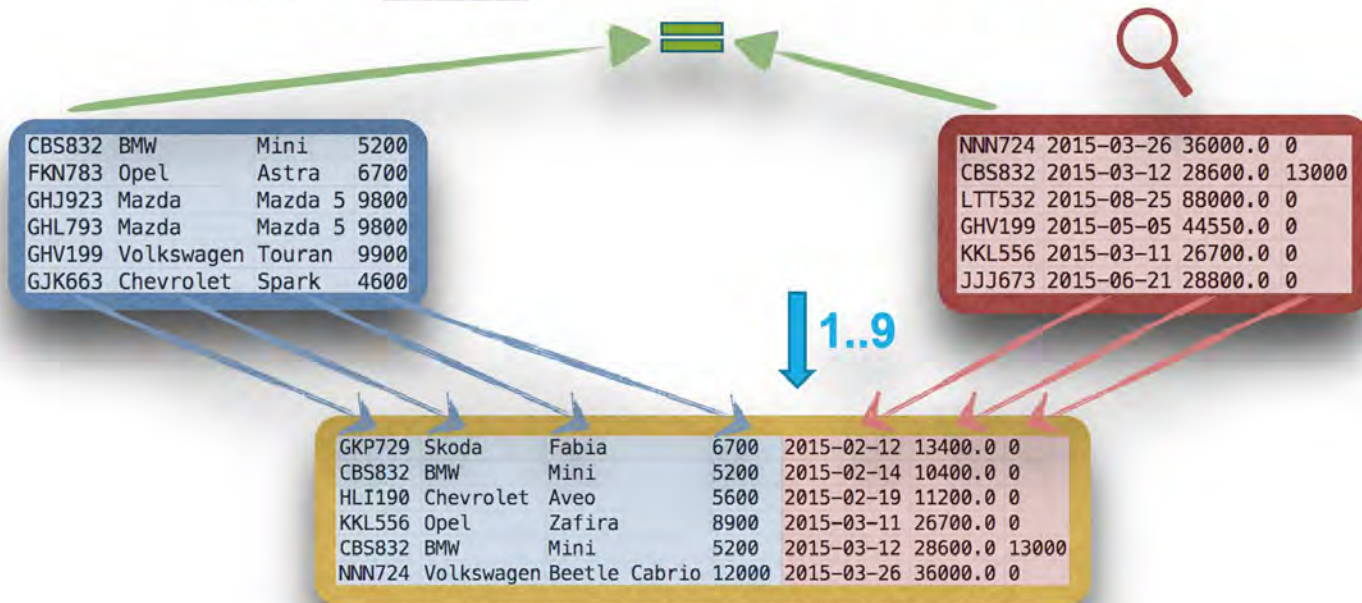
NNN724	2015-03-26	36000.0	0
CBS832	2015-03-12	28600.0	13000
LTT532	2015-08-25	88000.0	0
GHV199	2015-05-05	44550.0	0
KKL556	2015-03-11	26700.0	0
JJJ673	2015-06-21	28800.0	0

```
SELECT car.NPL, car.Maker, car.Model, car.price,
       rent.start, rent.rentprice, rent.exprice
FROM car
JOIN rent ON car.NPL = rent.NPL
WHERE rent.start BETWEEN '2015-02-01' AND '2015-08-31'
ORDER BY rent.start
```

GKP729	Skoda	Fabia	6700	2015-02-12	13400.0	0
CBS832	BMW	Mini	5200	2015-02-14	10400.0	0
HLI190	Chevrolet	Aveo	5600	2015-02-19	11200.0	0
KKL556	Opel	Zafira	8900	2015-03-11	26700.0	0
CBS832	BMW	Mini	5200	2015-03-12	28600.0	13000
NNN724	Volkswagen	Beetle Cabrio	12000	2015-03-26	36000.0	0

JOIN IV.

```
SELECT car.NPL, car.Maker, car.Model, car.price,  
       rent.start, rent.rentprice, rent.exprice  
FROM car  
JOIN rent ON car.NPL = rent.NPL  
WHERE rent.start BETWEEN '2015-02-01' AND '2015-08-31'  
ORDER BY rent.start
```



Summary – Queries I.

- * = record and not all attributes!
- Calculated attributes, **DISTINCT**, **TOP**
- Simple and complex conditions

BETWEEN	NOT BETWEEN
LIKE	NOT LIKE
IS NULL	IS NOT NULL
IN	NOT IN
EXISTS	NOT EXISTS

- Result set ordering
ORDER BY [OFFSET ... ROWS FETCH]
- Join the tables
JOIN
{ LEFT | RIGHT | INNER | OUTER | CROSS } JOIN

Summary – Queries II.

- **Grouping and aggregating**

- aggregating functions: **COUNT, SUM, MIN, MAX, AVG, STRING_AGG**
- Modifying effect of **DISTINCT**
- **GROUP BY**
 - The group attribute may exists in the **SELECT** header.
- **HAVING**

	WHERE	HAVING
When?	Before grouping	After grouping
What?	Records	Groups
Aggregation function allowed	X	✓

- **Embedded queries**

- **Usage of IN, ANY, ALL**
- **Embedding**
 - In the attribute list (**SELECT**),
 - In the tables (**FROM**),
 - In the conditions (**WHERE**),
 - In the group conditions (**HAVING**)

- **Set operators**

- **UNION, UNION ALL, INTERSECT, EXCEPT**

Data definition and data modification language

Create and drop tables

- CREATE TABLE

```
CREATE TABLE table_name  
  (column_name1 data_type [constraints],  
   column_name2 data_type [constraints],  
   column_name3 data_type [constraints]  
   [integrity-constraintk]);
```

- DROP TABLE

```
DROP TABLE [IF EXISTS] table_name;
```

Create and drop tables

- *Create the CLASS table:*
 - **CLASS**(ClassID, Class)
 - **ClassID**: primary key, auto increment
 - **Class**: unique, null is not allowed

Initial value

Increment

```
CREATE TABLE Class (  
    ClassID INT IDENTITY (1, 1) PRIMARY KEY,  
    Class VARCHAR(50) UNIQUE NOT NULL  
)
```

Note: The UNIQUE constraint is created as an index

Create and drop tables

- *Create the EMPLOYEE table:*
 - EMPLOYEE(EmployeeID, Name, Birthday, Sex, Salary, Premium, ClassID)
 - **EmployeeID:** primary key, auto increment
 - **Name:** null is not allowed
 - **Birthday:** date
 - **Sex:** default value is 'M'
 - **Salary:** greater than 3 000
 - **Premium:** Salary x 0.1
 - **ClassID:** foreign key for table CLASS and column *ClassID*

```
CREATE TABLE Employee (  
    EmployeeID INT IDENTITY(1,1) PRIMARY KEY,  
    Name VARCHAR(50) NOT NULL,  
    Birthday DATE,  
    Sex CHAR(1) DEFAULT 'M',  
    Salary INT CHECK (Salary > 3000),  
    Premium AS Salary * 0.1, -- PERSISTED  
    ClassID INT REFERENCES Class  
        ON UPDATE CASCADE  
        ON DELETE SET NULL  
)
```

Modify table

- **ALTER TABLE**

- Add a new column

```
ALTER TABLE table_name  
ADD column_name datatype
```

- Drop a column

```
ALTER TABLE table_name  
DROP COLUMN column_name
```

- Modify a type of column

```
ALTER TABLE table_name  
ALTER COLUMN column_name datatype
```


Add and remove a constraint

- **ALTER TABLE**

- Add a new constraint

ALTER TABLE table_name

ADD CONSTRAINT constraint_name *constraint_definition*

- Drop a constraint

ALTER TABLE table_name

DROP CONSTRAINT constraint_name

Practice

- *Add an integer column named Boss to the CLASS table!*

```
ALTER TABLE Class
```

```
ADD Boss INT
```

- *Refer to the primary key of the EMPLOYEE table!*

```
ALTER TABLE Class
```

```
ADD CONSTRAINT fk_boss FOREIGN KEY (Boss) REFERENCES Employee (EmployeeID)
```

- *The previous two commands in one statement:*

```
ALTER TABLE Class
```

```
ADD Boss INT REFERENCES Employee
```

- *Drop the minimum payment limit!*

```
ALTER TABLE Employee
```

```
DROP CONSTRAINT CK__employee__salary__7CF981FA
```

User defined types

- **CREATE TYPE - DROP TYPE**
 - It can be created from a basic data type
 - For example, it is good for enumerations (there is not ENUM type in Microsoft SQL)
 - It can be an embedded table

Rules

- **Implementation of complex conditions**
 - Creating rules
 - It can be attached to several columns
 - Examples: Phone number, name or address format, numeric values delimitation

<https://www.simple-talk.com/sql/learn-sql-server/robyn-pages-sql-server-data-validation-workbench/>

Data type and rule example

- *Create a USERNAME data type which cannot contain the following values: 'admin', 'administrator', 'guest'!*

```
CREATE TYPE USERNAME FROM VARCHAR(20)
GO
```

```
CREATE RULE username_domain AS
    @username = TRIM(@username) AND
    LOWER(@username) NOT IN ('admin', 'administrator', 'guest')
GO
```

```
EXEC sp_bindrule 'username_domain', 'username'
GO
```

```
CREATE TABLE usertable (
    id INT IDENTITY (1, 1) PRIMARY KEY,
    name USERNAME NOT NULL
)
```


Create and drop an index

- CREATE INDEX
 - **CREATE [UNIQUE][CLUSTERED | NONCLUSTERED] INDEX index_name
ON table_name (column_name[ASC | DESC]) [INCLUDE (column_name)]**
- DROP INDEX
 - **DROP INDEX [IF EXISTS] index_name ON table_name**

Index – Practice

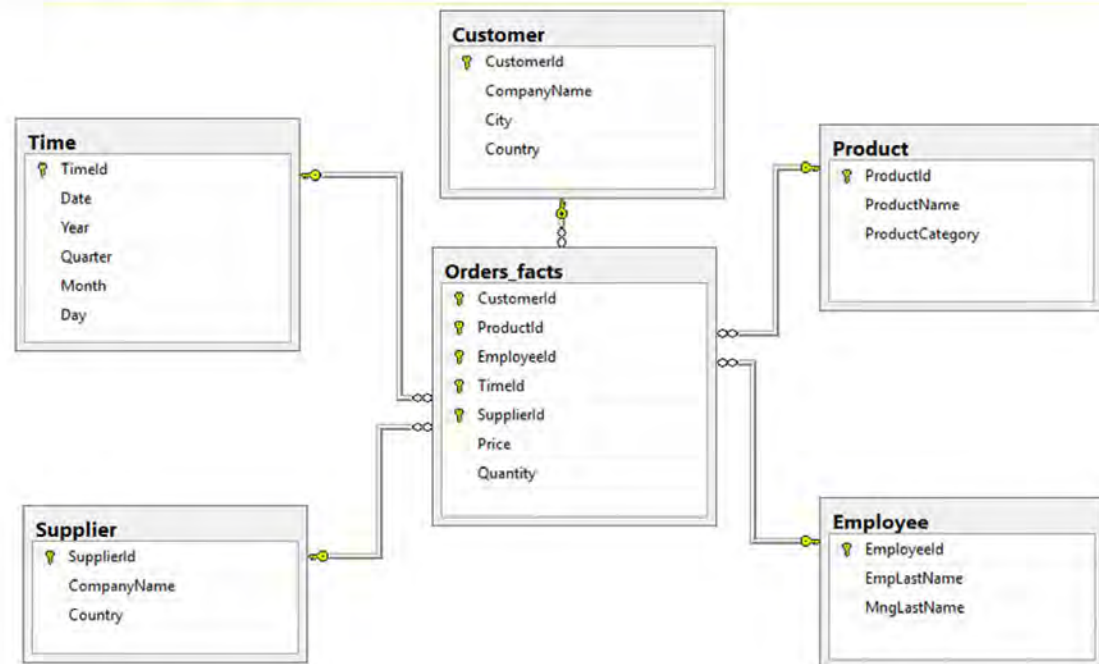
- *Create a non-clustered index for the Name column of the EMPLOYEE table, and then drop it!*

```
CREATE NONCLUSTERED INDEX nameindex  
ON Employee (Name)
```

```
DROP INDEX nameindex  
ON Employee
```

Database diagram

- Graphically display the data connections
- Can be generated from existing tables
- Suitable for defining foreign keys



Data export and import

- Db / Tasks / Export Data and Import Data
- Source and destination files or databases:
 - Both must exist!
- Options
 - Character encoding
 - Number and date formats
 - String delimiters

Insert new record(s)

- GUI interface
- Using SQL script
 - **INSERT INTO ... VALUES**
 - INSERT INTO TableName [(columnList)]
VALUES (dataValueList)
 - **INSERT INTO ... SELECT**
 - INSERT INTO TableName [(columnList)]
SELECT ...
 - **SELECT INTO ... FROM...**
 - SELECT columnList INTO newTable
FROM Table
[WHERE ...]

columnList is optional – if not exists -> all columns
dataValueList and columnList

- have the same number
- have the same position
- pairs of data types are compatible

Insert new record(s) – Practice

- *Let's append a new record into a few tables and observe! What happened with*
 - Default values: automatically entered
 - Calculated field: automatically calculated, when all source values are available
 - Assign to Identity field: Not allowed!
 - Assigning a bit field: 0 or 1 (SSMS: true or false)

INSERT

INTO car

(NPL, Maker, Model, Price, Fuel, Persons, Available)

VALUES ('AAA111', 'Trabant', 'Combi', 1000, 'petrol', 4, 1)

Insert new record(s) – Practice

- *Add multiple records to the CLIENT table at the same time (with values in name, birth, sex and countryID columns)!*
- *In the sex column enter the default value at first time and leave it blank the next.*

```
INSERT
  INTO client
  (name, birth, sex, countryID)
VALUES ('Somebody1', '1980-01-01', DEFAULT, 'HUN'),
       ('Somebody2', '1980-01-01', NULL, 'HUN')
```

Insert new record(s) – INSERT FROM

- *Create a table named COUNTRY2 and copy the records of the COUNTRY table to it! Then drop the COUNTRY2 table!*
- Create the table

```
CREATE TABLE Country2 (  
    countryID CHAR(3) NOT NULL PRIMARY KEY,  
    country VARCHAR(15)  
)
```

- Copy data

```
INSERT  
    INTO Country2  
    SELECT *  
    FROM Country
```

- Delete table

```
DROP TABLE Country2
```

Insert new record(s) – SELECT INTO

- *Copy the cars for people 5 (maximum number of persons that can be carried) from table CAR into CAR5 in such a way, that CAR5 table is not created previously! In the new table, the records should be sorted by number plate! At the end of the operation, drop the new table!*

```
SELECT *  
  INTO car5  
  FROM car  
 WHERE persons = 5  
 ORDER BY NPL
```

```
DROP TABLE car5
```

Modification of data

- GUI interface
- Using SQL script
 - **UPDATE** [**TOP** (expression) **PERCENT**] TableName
 SET columnName1 = dataValue1
 [, columnName2 = dataValue2...]
 [**WHERE** searchCondition]
 - **TableName:** Table or modifiable view
 - **SET:** Columns to change
 - The dataValue values must be type compatible
 - **WHERE:** Optional
 - If not, all records are modified

Modification of data – Practice

- *Change the price of car 'CBS832' to 6,200!*

```
UPDATE car
  SET price = 6200
  WHERE NPL = 'CBS832'
```

- *Decrease the price of 3 random cars by 10%!*

```
UPDATE TOP (3) car
  SET price = price * 0.9
```

- *Decrease the price of 3 most expensive cars by 10%!*

```
UPDATE car
  SET price = price * 0.9
  WHERE NPL IN (SELECT TOP (3) WITH TIES NPL
                FROM car
                ORDER BY price DESC)
```

Modification of data – UPDATE-FROM

- *Temporarily modify the CLIENT table to include a 'country' column and update the values for this column based on the COUNTRY table! Finally, delete this field from the CLIENT table!*

```
ALTER TABLE client
ADD country VARCHAR(15)
```

```
UPDATE client
SET country = o.country
FROM client AS u
JOIN country AS o ON u.countryID = o.countryID
```

```
ALTER TABLE client
DROP COLUMN country
```

Modification of data – Practice

- *Add a column called times to CLIENT table and enter the number of times each client rented (based on the data in the RENT table)!*

```
ALTER TABLE client
ADD number_of_loans INT
```

```
UPDATE client
SET number_of_loans = r.number_of_loans
FROM client AS u
JOIN (SELECT clientID, COUNT(*) AS number_of_loans
      FROM rent
      GROUP BY clientID) AS r
ON u.clientID = r.clientID
```

```
UPDATE client
SET number_of_loans = 0
WHERE number_of_loans IS NULL
```

Delete record(s)

- GUI interface
- Using SQL script
 - **DELETE [TOP (expression) PERCENT]**
FROM TableName
[WHERE searchCondition]
 - TableName: Table or modifiable view
 - WHERE: Optional
 - If not, all records are deleted
 - **TRUNCATE TABLE** TableName
 - Removes all rows from a table or specified partitions of a table, without logging the individual row deletions.
 - Cannot use truncate on tables that:
 - are referenced by a foreign key, participate in an indexed view, are published by using transactional replication or merge replication

Delete record(s) – Practice

- *Delete clients born after January 1, 2000!*

```
DELETE FROM client  
WHERE birth >= '2000-01-01'
```


MERGE – “Synchronize” records

- Inserts, modifies, or deletes existing records, depending on whether the specified condition is met.

```
MERGE [INTO] target_table USING table_source
ON (condition)
  WHEN MATCHED THEN
    ...
  WHEN NOT MATCHED [BY TARGET] THEN
    ...
  WHEN NOT MATCHED BY SOURCE THEN
    ...
```

Practice I.

- *Duplicate the CAR table as CAR2, then modify and delete some records. Additionally, add a new car to the CAR2 table!*

```
SELECT *  
  INTO car2  
  FROM car
```

Practice II.

- *Synchronize the CAR2 table based on the CAR table using the MERGE statement!*
- *Delete unnecessary records:*

1.

```
DELETE
  FROM car2
  WHERE NLP NOT IN (SELECT NLP
                    FROM car)
```
2.

```
DELETE car2
  FROM car2
  LEFT OUTER JOIN car ON car2.NLP = car.NLP
  WHERE car.NLP IS NULL
```
3.

```
DELETE
  FROM car2
  WHERE NOT EXISTS (SELECT NLP
                   FROM car
                   WHERE car.NLP = car2.NLP)
```

Practice III.

- To **add**, **modify**, and **delete** records from the CAR table:

```
MERGE INTO car2 AS target
  USING car AS source
  ON (target.NLP = source.NLP)
  WHEN MATCHED THEN
    UPDATE
      SET maker      = source.maker,
          model      = source.model,
          [year]     = source.[year],
          price      = source.price,
          fuel       = source.fuel,
          persons    = source.persons,
          available  = source.available
  WHEN NOT MATCHED THEN
    INSERT (NLP, maker, model, [year], price, fuel,
           persons, available)
    VALUES (source.NLP, source.maker, source.model,
            source.[year], source.price, source.fuel,
            source.persons, source.available)
  WHEN NOT MATCHED BY SOURCE THEN
    DELETE ;
```



Practice IV.

- To **add**, and **modify** records from the CAR table:

```
MERGE INTO car2 AS target
  USING car AS source
  ON (target.NLP = source.NLP)
  WHEN MATCHED THEN
    UPDATE
      SET maker      = source.maker,
          model      = source.model,
          [year]     = source.[year],
          price      = source.price,
          fuel       = source.fuel,
          persons    = source.persons,
          available  = source.available
  WHEN NOT MATCHED THEN
    INSERT (NLP, maker, model, [year], price, fuel,
            persons, available)
    VALUES (source.NLP, source.maker, source.model,
            source.[year], source.price, source.fuel,
            source.persons, source.available);
```



Views

- Virtual table
 - **CREATE VIEW** [schema_name.]view_name [(**column** [,...n])]
[**WITH** <view_attribute> [,...n]]
AS select_statement
[**WITH CHECK OPTION**] [;]
 - **DROP VIEW** [**IF EXISTS**] view_name [;]
- The SELECT clauses in a view definition cannot include the following:
 - An ORDER BY clause, unless there is also a TOP clause in the select list of the SELECT statement
 - The INTO keyword
 - The OPTION clause
 - A reference to a temporary table or a table variable
- Data manipulation operations may be also performed via views.

Views – Practice

- *Create a view for the 3 most expensive cars!*

```
CREATE VIEW vMEcars AS  
SELECT TOP (3) *  
FROM car  
ORDER BY Price DESC
```

- *Drop the vMEcars view if exists!*

```
DROP VIEW IF EXISTS vMEcars
```

CASE in the SELECT statement and Common Table Expressions

Conditional structure, branches

- Simple CASE expression $\approx$ switch - case

```
CASE input_expression  
  WHEN when_expression THEN result_expression  
  [ ...n ]  
  [ ELSE else_result_expression ]  
END
```

- Searched CASE expression $\approx$ if - then - else

```
CASE  
  WHEN Boolean_expression THEN result_expression  
  [ ...n ]  
  [ ELSE else_result_expression ]  
END
```

CASE – Practice 1.

- Display the data in the CLIENT table with the name and sex (displayed as 'male' and 'female') of the client.

```
SELECT name,  
       CASE  
         WHEN sex = 'f' THEN 'female'  
         WHEN sex = 'm' THEN 'women'  
         ELSE 'n.a.'  
       END AS sex  
FROM client
```

Searched expression

```
SELECT name,  
       CASE sex  
         WHEN 'f' THEN 'female'  
         WHEN 'm' THEN 'women'  
         ELSE 'n.a.'  
       END AS sex  
FROM client
```

Simple expression

CASE – Practice 2.

- *List the number of rentals on a monthly basis!*

```
SELECT CONCAT(YEAR(startDate), '.', MONTH(startDate)) AS [Month],  
        COUNT(rentID) AS [Number of Rents]  
FROM rent  
GROUP BY CONCAT(YEAR(startDate), '.', MONTH(startDate))
```

```
SELECT month AS [Month], COUNT(rentID) AS [Number of Rents]  
FROM (  
    SELECT CONCAT(YEAR(startDate), '.', MONTH(startDate)) AS month,  
           rentID  
    FROM rent  
    ) AS x  
GROUP BY month
```

CASE – Practice 3.

- *List the number of borrowings on a monthly basis in such a way, that the dates of the rentals are always displayed in the format of YYYY.MM!*

```
SELECT CONCAT(YEAR(startDate), '.',  
             CASE WHEN MONTH(startDate) < 10 THEN '0' END,  
             MONTH(startDate)) AS [Month],  
       COUNT(rentID) AS [Number of Rents]  
FROM rent  
GROUP BY CONCAT(YEAR(startDate), '.',  
                CASE WHEN MONTH(startDate) < 10 THEN '0' END,  
                MONTH(startDate))
```

```
SELECT [Month], COUNT(rentID) AS [Number of Rents]  
FROM (SELECT CONCAT(YEAR(startDate), '.',  
                   CASE WHEN MONTH(startDate) < 10 THEN '0' END,  
                   MONTH(startDate)) AS [Month],  
          rentID  
       FROM rent) AS x  
GROUP BY [Month]
```

CASE – Practice 3.

- *Increase the price of Volkswagen cars by 20% and from all other manufacturers by 10%! Solve the task with a single statement!*

```
UPDATE car
  SET price = price *
    CASE
      WHEN Maker = 'Volkswagen' THEN 1.2
      ELSE 1.1
    END
```

CTE - Common Table Expression

- Temporary result-set

```
[ WITH <common_table_expression> [ ,...n ] ]
```

```
<common_table_expression>::=
```

```
expression_name [ ( column_name [ ,...n ] ) ]
```

```
AS
```

```
( CTE_query_definition )
```

```
WITH cteName [ ( column_name [ ,...n ] ) ] AS
```

```
(
```

```
<Inner query definition>
```

```
SELECT ...
```

```
FROM ...
```

```
)
```

```
<Outer query definition>
```

```
SELECT ...
```

```
FROM cteName
```

<https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql?view=sql-server-2017>

CTE – Practice 1.

- *Show all Mazda cars for which the daily rental price is less than the average rental price of Mazdas!*

```
SELECT *  
FROM car  
WHERE maker = 'Mazda'  
AND price < (SELECT AVG(price)  
              FROM car  
              WHERE maker = 'Mazda')
```

CTE

```
WITH mazda AS (  
    SELECT maker, price  
    FROM car  
    WHERE maker = 'Mazda'  
)  
SELECT *  
FROM mazda  
WHERE price < (SELECT AVG(price)  
               FROM mazda)
```


CTE – Practice 2.

- *Expand the previous task so that only five cars are included in the query!*

```
SELECT *
FROM car
WHERE maker = 'Mazda'
      AND persons = 5
      AND price < (SELECT AVG(price)
                   FROM car
                   WHERE maker = 'Mazda'
                   AND persons = 5)
```

```
WITH mazda5 AS (
  SELECT maker, price
  FROM car
  WHERE maker = 'Mazda'
        AND persons = 5
)
SELECT *
FROM mazda5
WHERE price < (SELECT AVG(price)
               FROM mazda5)
```

```
-- Define the CTE expression name
WITH mazda AS (
-- Define the CTE query
  SELECT maker, price
  FROM car
  WHERE maker = 'Mazda'
), -- Use a comma to separate multiple CTE definitions
  mazda5 AS (
  SELECT maker, price
  FROM mazda
  WHERE persons = 5
)
-- Define the outer query referencing the CTE name
SELECT *
FROM mazda5
WHERE price < (SELECT AVG(price)
               FROM mazda5)
```

CTE – Practice 2.

- Recursive query using CTE

```
WITH boss (bossID, employeeID, employeeName, level) AS (
```

```
    -- "Define the first CTE query."
```

```
    SELECT ReportsTo AS bossID, EmployeeID AS employeeID,  
           CONCAT(FirstName, ' ', LastName) AS employeeName,  
           0 AS level  
    FROM Employees  
    WHERE ReportsTo IS NULL
```

```
    UNION ALL
```

```
    -- Recursive query
```

```
    SELECT ReportsTo, EmployeeID, CONCAT(FirstName, ' ', LastName), level + 1  
    FROM Employees AS e  
    JOIN boss AS f ON e.ReportsTo = f.employeeID
```

```
)
```

```
    -- Query
```

```
SELECT *  
FROM boss
```



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének
együttes javítása a Pannon Egyetemen

MS SQL 2019+ PRACTICE

Slides was compiled by
István Vassányi, Károly Machalik

SZÉCHENYI 2020



MAGYARORSZÁG
KORMÁNYA

Európai Unió
Európai Strukturális
és Beruházási Alapok



BEFEKTETÉS A JÖVŐBE

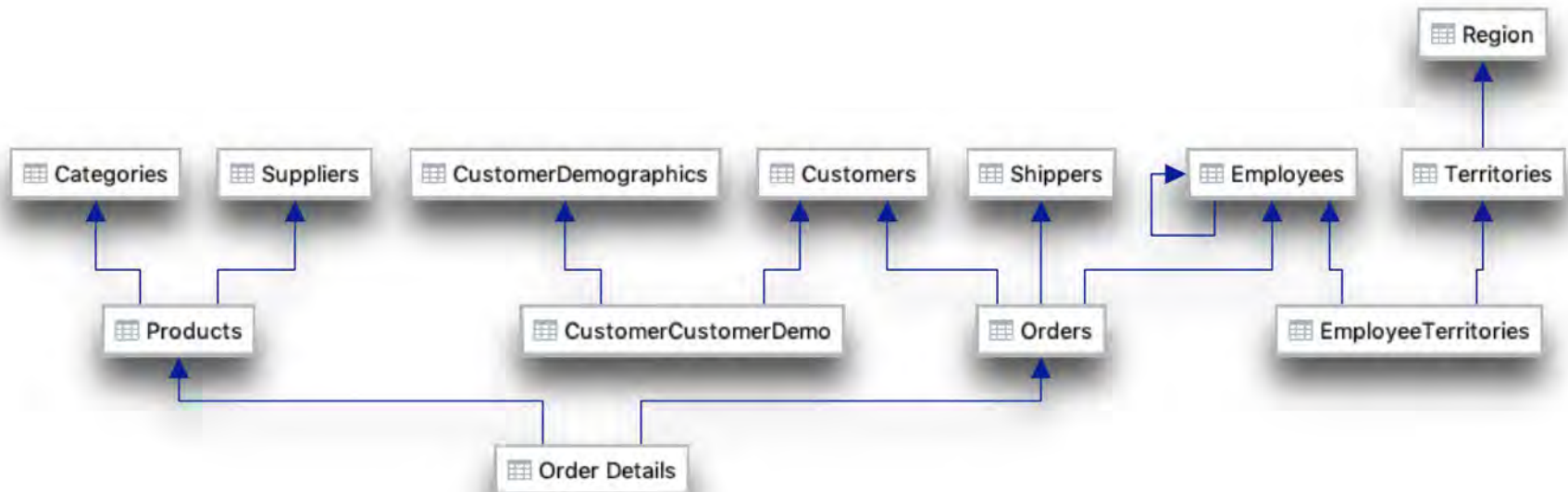
Northwind database

Introduction – Northwind database

- For a part of the examples, a database of a commercial company is used
- This database is freely available and the SQL script needed to create it can be downloaded from GitHub
- The **Employees** table contains the company's employees. Most agents who are responsible for a particular area. Areas belonging to an agent are shown on the **EmployeeTerritories** table. The agents concluded orders (**Orders** table) with customers (**Customers** table). The order includes one or more items (**Order Details** table). The item includes the ordered product, its quantity, the unit price at the time of order and the discount. Orders are shipped by courier companies, which are included in the **Shippers** table. Orders are delivered from the company's warehouse, the current inventory of which is included in the **Products** table. All products are purchased from one supplier (**Suppliers** table).

<https://github.com/Microsoft/sql-server-samples/tree/master/samples/databases/northwind-pubs>

Introduction – Northwind database



Introduction – Northwind database

- **Categories**

- CategoryID
- CategoryName

Product categories

Category identity, primary key

Category name, i.e. “Beverages”

Categories	
 CategoryID	int
 CategoryName	nvarchar(15)
 Description	ntext
 Picture	image

Introduction – Northwind database

- **Products**

- ProductID
- SupplierID
- CategoryID
- ProductName
- UnitPrice
- UnitsInStock
- ReorderLevel
- UnitsOnOrder
- Discontinued

Products

Product identity, primary key

Supplier identity, foreign key to **Supplier**

Product category identity, foreign key to **Categories**

Product name

Current unit price of product

Current stock level of product

Minimal stock level for reorder

The number of units on the supplier order
(not yet received)

Active (0) or discontinued (1) product

Products	
 ProductID	int
 ProductName	nvarchar(40)
 SupplierID	int
 CategoryID	int
 QuantityPerUnit	nvarchar(20)
 UnitPrice	money
 UnitsInStock	smallint
 UnitsOnOrder	smallint
 ReorderLevel	smallint
 Discontinued	bit

Introduction – Northwind database

- **Orders**

- OrderID
- OrderDate
- EmployeeID
- CustomerID
- ShipVia

Orders

Order identity, primary key

Order date

Employee identity, foreign key to **Employee**

Customer identity, foreign key to **Customers**

Shipper identity, foreign key to **Shippers**

Orders	
 OrderID	int
 CustomerID	nchar(5)
 EmployeeID	int
 OrderDate	datetime
 RequiredDate	datetime
 ShippedDate	datetime
 ShipVia	int
 Freight	money
 ShipName	nvarchar(40)
 ShipAddress	nvarchar(60)
 ShipCity	nvarchar(15)
 ShipRegion	nvarchar(15)
 ShipPostalCode	nvarchar(10)
 ShipCountry	nvarchar(15)

Introduction – Northwind database

- **Order details**

- OrderID, ProductID
- UnitPrice:
- Quantity
- Discount

Order details

A two-field key, and a separate foreign key for **Orders** and **Products**

The unit price for the item at the time of order (it may already be different in the **Products** table!)

Quantity

Discount - for example, if it is 0.05, the item value (UnitPrice * Quantity) should be reduced by 5%

Order Details	
 OrderID	int
 ProductID	int
 UnitPrice	money
 Quantity	smallint
 Discount	real

Introduction – Northwind database

- **Employees**

	Employees
▪ <u>EmployeeID</u>	Employee identity, primary key
▪ LastName	Last name
▪ FirstName	First name
▪ Title	Title, such as “Sales Manager”
▪ Country, City	Address of the employee
▪ ReportsTo	Boss of employee, foreign key to Employee

Employees	
EmployeeID	int
LastName	nvarchar(20)
FirstName	nvarchar(10)
Title	nvarchar(30)
TitleOfCourtesy	nvarchar(25)
BirthDate	datetime
HireDate	datetime
Address	nvarchar(60)
City	nvarchar(15)
Region	nvarchar(15)
PostalCode	nvarchar(10)
Country	nvarchar(15)
HomePhone	nvarchar(24)
Extension	nvarchar(4)
Photo	image
Notes	ntext
ReportsTo	int
PhotoPath	nvarchar(255)

Introduction – Northwind database

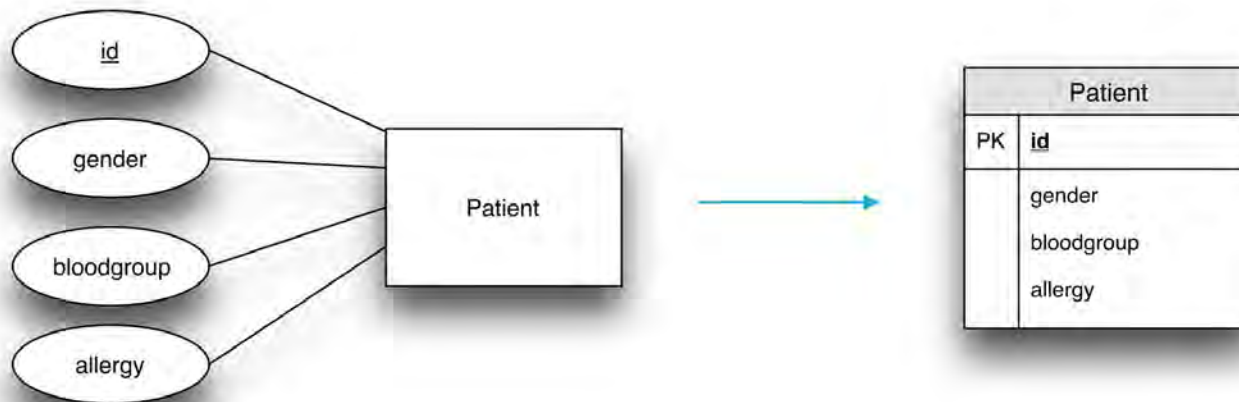
- **EmployeeTerritories** **Operational territories of employees**
 - EmployeeID Foreign key to **Employee**
 - TerritoryID Foreign key to **Territories**

EmployeeTerritories	
EmployeeID	int
TerritoryID	nvarchar(20)

Data modeling

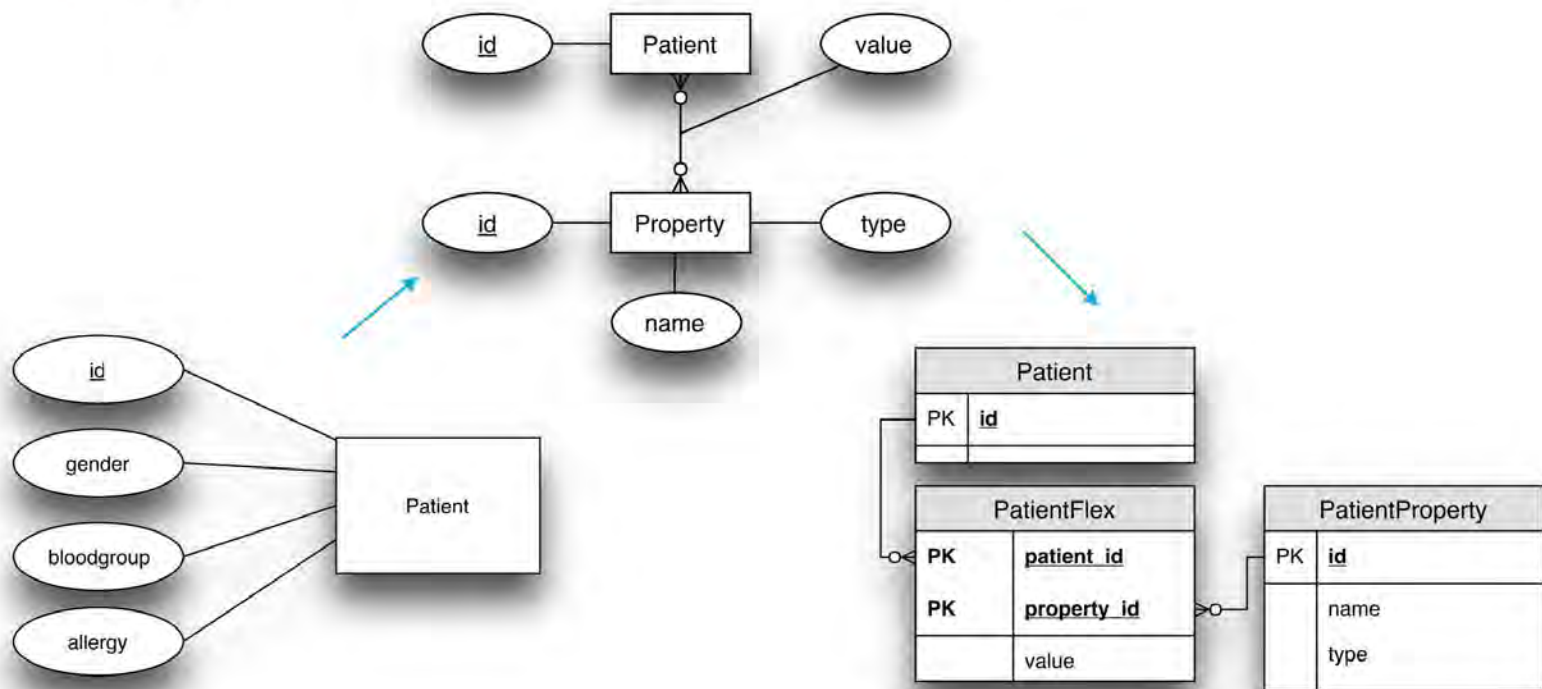
Data modeling

- Field-oriented modeling
 - Each attribute in the object has one field
 - + Easily programmable, efficient GUI creation, automated ORM environment
 - - Difficult to change, the modification is costly and dangerous



Data modeling

- Record-oriented modeling
 - Each attribute in the object has one record
 - + Flexible, easy-to-change structure
 - Attributes belonging to an object should be retrieved from multiple tables



Data modeling – Field ↔ Record

	Flexibility	Programmability	Efficiency
Field-oriented	Rigid, to add a new attribute, you need to modify the data model and the underlying logic.	Easily and quickly be programmed. Most GUI developer and ORM environments offer automated solutions for linking fields to attributes or GUI elements (such as text boxes) - often without programming.	Also stores blank (NULL) fields.*
Record-oriented	Flexible, durable. To add a new attribute, it is enough to add a new record to the attribute master table.	Cumbersome. Attributes belonging to a single instance of objects must be retrieved from multiple records. An interface layer or separate GUI programming is required.	Improved efficiency: does not store unknown attribute values.

Data modeling – Field $\leftrightarrow$ Record

Field-oriented

Patient

id	gender	bloodgroup	allergy
100	Female	NULL	Lidocain
101	Male	A+	NULL

Record-oriented

Patientx

id
100
101

PatientFlex

patient_id	property_id	value
100	1	Female
100	3	Lidocain
101	1	Male
101	2	A+

PatientProperty

id	name	type
1	gender	enum
2	blood group	enum
3	allergies	coded list

- ✦ The aggregation of data sets from different data sources is also called a data federation. A typical task of the data federation is to transform the record-oriented model into a field-oriented model and vice versa.

Data modeling – Field ↔ Record

- Copy the data from the **Employees** table in the **Northwind** database to your own database, called **EmployeeField**. Copy only the *EmployeeID*, *LastName*, *Title*, *City* attributes!

```
SELECT EmployeeID, LastName, Title, City
INTO EmployeeField
FROM Northwind..Employees
```

- Create an **Attributes** table as follows:

<u>AttribId</u>	integer, primary key
AttribName	nvarchar, maximum 100 character
AttribType	nvarchar, maximum 100 character

```
CREATE TABLE Attributes (
    AttribId INT PRIMARY KEY,
    AttribName NVARCHAR(100),
    AttribType NVARCHAR(100)
)
```

Data modeling – Field ↔ Record

- Fill in the **Attributes** table as follows!

```
INSERT INTO Attributes (AttribId, AttribName, AttribType)
VALUES (1, 'Last name', 'text'),
       (2, 'Title', 'text'),
       (3, 'City', 'text')
```

AttribId	AttribName	AttribType
1	Last name	text
2	Title	text
3	City	text

- Create an **EmployeeRecord** table as follows:
 - **EmployeeId** integer, NULL not allowed
 - **AttribId** integer, NULL not allowed, foreign key to Attributes table
 - **AttribValue** nvarchar, maximum 500 character

```
CREATE TABLE EmployeeRecord (
    EmployeeId INT NOT NULL,
    AttribId INT NOT NULL REFERENCES Attributes,
    AttribValue NVARCHAR(500)
    PRIMARY KEY (EmployeeId, AttribId)
)
```


Data modeling – Field ↔ Record

- Fill in the **EmployeeRecords** table with **EmployeeField** data!

```
INSERT INTO EmployeeRecord (EmployeeId, AttribId, AttribValue)
  SELECT EmployeeID, 1, LastName
    FROM EmployeeField
    WHERE LastName IS NOT NULL
UNION
  SELECT EmployeeID, 2, Title
    FROM EmployeeField
    WHERE Title IS NOT NULL
UNION
  SELECT EmployeeID, 3, City
    FROM EmployeeField
    WHERE City IS NOT NULL
```

- Verify that the data has been uploaded!

```
SELECT *
  FROM EmployeeRecord
```


Data modeling – Field ↔ Record

- Create a query that displays the data and its type and name using the **EmployeeRecord** and **Attributes** tables.

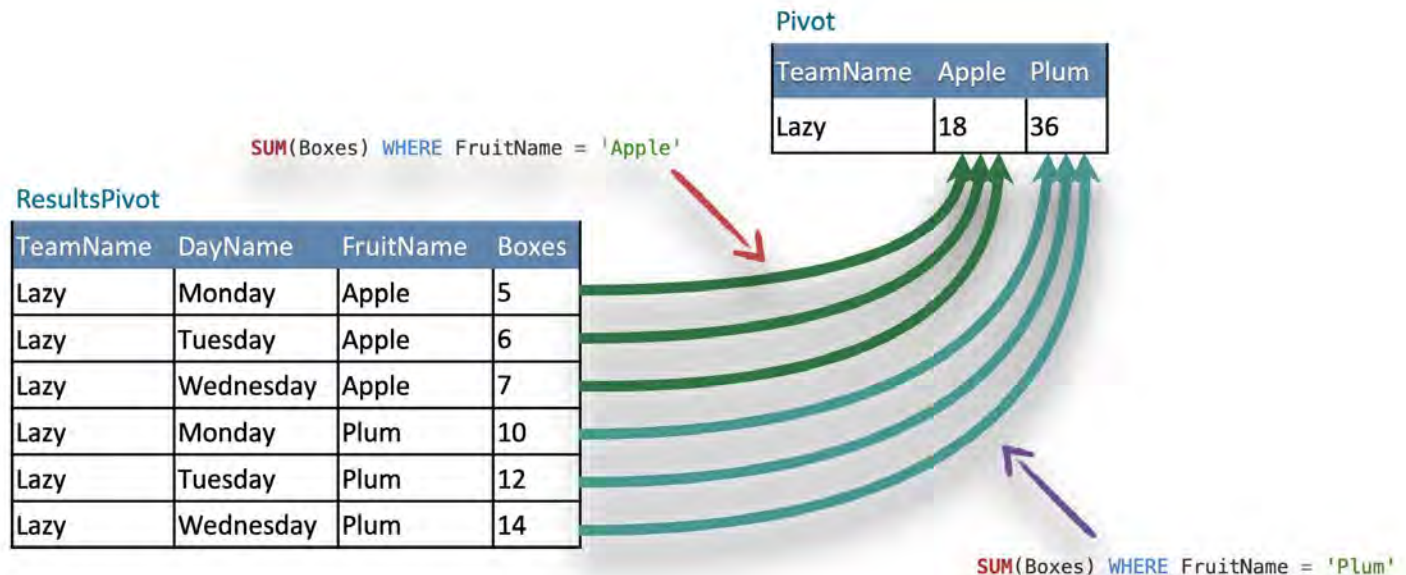
```
SELECT e.EmployeeId, a.AttribName, a.AttribType, e.AttribValue
FROM EmployeeRecord AS e
JOIN Attributes AS a ON e.AttribId = a.AttribId
```

- Create a query that displays the **EmployeeRecord** table data according to the original format!

```
SELECT EmployeeId,
       MIN(CASE WHEN AttribId = 1 THEN AttribValue END) AS LastName,
       MIN(CASE WHEN AttribId = 2 THEN AttribValue END) AS Title,
       MIN(CASE WHEN AttribId = 3 THEN AttribValue END) AS City
FROM EmployeeRecord
GROUP BY EmployeeId
```

Data modeling – PIVOT/UNPIVOT

- The original meaning of the English pivot word is tilt corner, tilting. In the case of a database, different dimensions can be transposed. Various aggregating functions can be used during transposition.



Data modeling – PIVOT/UNPIVOT

- Syntax of PIVOT operator

```
SELECT <non-pivoted column>,  
      [1st pivoted column] AS <column name>, [2nd pivoted column] AS <column name>, ...  
      [last pivoted column] AS <column name>  
FROM (<SELECT query that produces the data>) AS <alias for the source query>  
PIVOT (<aggregation function>(<column being aggregated>)  
      FOR [<column that contains the values that will become column headers>]  
      IN ( [first pivoted column], [second pivoted column], ... [last pivoted column])  
      ) AS <alias for the pivot table>  
<optional ORDER BY clause>;
```

<https://docs.microsoft.com/en-us/sql/t-sql/queries/from-using-pivot-and-unpivot?view=sql-server-2017>

Data modeling – PIVOT/UNPIVOT

- Create a query that displays the performance of the teams as fruit on the basis of the **ResultsPivot** table data. Show the names and results of the teams in the rows, and show the fruits in the columns!

```
SELECT p.TeamName, p.Apple, p.Plum
FROM (SELECT TeamName, FruitName, SUM(Boxes) AS Boxes
      FROM ResultsPivot
      GROUP BY TeamName, FruitName) AS s
PIVOT (SUM(Boxes) FOR FruitName IN ([Apple], [Plum])) AS p
```

- Create a query that displays the performance of the teams as fruit on the basis of the **ResultsPivot** table data. The rows show the names of the fruits and the results of the teams, while the columns show the names of the teams.

```
SELECT p.FruitName, p.Lazy, p.Diligent
FROM (SELECT TeamName, FruitName, SUM(Boxes) AS Boxes
      FROM ResultsPivot
      GROUP BY TeamName, FruitName) AS s
PIVOT (SUM(Boxes) FOR TeamName IN ([Lazy], [Diligent])) AS p
```

T-SQL

Database programming in T-SQL language

- **The T-SQL script may contain other language elements besides the already known series of DDL and DML SQL statements:**
 - Declaration of variables
 - Cycles
 - Conditional branches
 - Error handling
 - etc.
- **Most SQL editors can use the GO keyword**
 - Separates the entire script into separate units (batch)
 - Running separate units is not related - except for the order
 - Not a language element, only the editor knows!

T-SQL – processing, properties

- **Steps to process T-SQL programs**
 - Syntax checking (parse)
 - Compiling
 - Executing
- **Typical for batches:**
 - In case of a syntax error, execution does not start (1. step, parse)
 - In the event of a runtime error, the part before the trigger-command runs
 - The variables are valid only within the given batch
 - If a referenced object does not exist, the run continues with the following statement
 - In case of a violation of constraints or an arithmetic error, the running of the current batch is interrupted

T-SQL- Batch

- Effects of different errors on running

```
CREATE TABLE example (  
    number INT  
)  
  
INSERT INTO example VALUES (44)  
GO
```

```
UPDATE example  
SET number = 21  
GO
```

```
SELEC *  
FROM example  
GO
```

```
SELECT *  
FROM example  
GO
```



Incorrect syntax near '*.

21

```
UPDATE example  
SET number = 22  
  
SELEC *  
FROM example  
  
UPDATE example  
SET number = 23  
GO
```

```
UPDATE example  
SET number = 22
```

```
SELECT *  
FROM exampl  
GO
```

```
SELECT *  
FROM example  
GO
```

]

]

Invalid object name 'exampl'.

22

T-SQL – Procedural extensions – Advantages

- **Depending on the server, many extensions are available, for example:**
 - Oracle PL/SQL
 - PostgreSQL PL/pgSQL, *Perl, *Python, *Java, *pgPSM, *php, *R, stb.
 - Microsoft SQL T-SQL, R, Python
- **The business logic layer is implemented on the database server**
- **Simplicity**
 - If all business logic is part of the database server, then a thin client is enough
- **Technological neutrality**
 - Stored procedures can be used from any software environment
- **Manageability**
 - There is no need to modify clients when changing the business logic.

T-SQL – Procedural extensions – Advantages

- **Data security**

- Clients do not have direct access to the database, it is sufficient to grant access to stored procedures. In the event of an application error or failure, the user will not have access to the data unauthorized.

- **Efficiency**

- There is no need to transfer large amounts of data, which means faster operation and less network traffic.

- **Transparency**

- Centralized coding and SQL language optimized for data management are more transparent than client-focused languages for other tasks.

T-SQL – Procedural extensions – Disadvantages

- **Low-level language**

- Procedural languages are often difficult to use for general programming tasks.

- **Software technology**

- Server-side support for software technology is usually not resolved. Debugging, VCS, teamwork, etc.

- **Scalability**

- Database server scaling is a routine task, but depending on the application, different resource optimization may be required, which can be very costly.

T-SQL – Procedural extensions

- Existence of specialized expertise for database administrators is essential for both server and client applications.
- In practice, for a particular application, it is usually not the ideology but the functionality that chooses the particular component to be located on the application or database server, or the client performs it.
- A script that creates a complete application database must be handled as the source code of the software. VCS, backup, etc.

T-SQL – Variables, flow control

- **DECLARE** Declaration of variables. The name begins with a @ character! It can also be used for assigning.
- **SET** Assignment
- **BEGIN...END** Command block
- **IF/ELSE** Conditional branching
- **WHILE/BREAK/CONTINUE** Front testing cycle. There is no other type of cycle.
- **RETURN** Exits the batch, block or stored procedure unconditionally.
- **TRY/CATCH/THROW** Exception handling (RAISERROR)
- **WAITFOR/DELAY/TIME** Waiting
- **GOTO** Go to a label
- **PRINT** Write text or variable value

T-SQL – Examples

- Create a script that displays the sum of numbers smaller than 50!

```
DECLARE @i INT = 1,  
        @result INT = 0
```

```
WHILE @i <= 50 BEGIN  
    SET @result += @i  
    SET @i += 1  
END
```

```
PRINT CONCAT('The sum of natural numbers less than 50: ', @result)  
GO
```

T-SQL – Examples

- Create a script that displays Fibonacci numbers below 1000!

```
DECLARE @i INT = 1,  
        @j INT = 1,  
        @temp INT
```

```
WHILE @i + @j < 1000 BEGIN  
    SET @temp = @j  
    SET @j = @j + @i  
    SET @i = @temp  
    PRINT @j  
END
```



```
2  
3  
5  
8  
13  
21  
34  
55  
89  
144  
233  
377  
610  
987
```


T-SQL – Examples

- Create a script that writes the title of a worker from the Employer's table in the Northwind database. (Country, city, address format)

```
USE northwind
```

```
DECLARE @name NVARCHAR(20) = 'Fuller%', -- King  
        @address NVARCHAR(max)
```

```
SELECT @address = CONCAT_WS(', ', Country, City, Address)  
FROM Employees  
WHERE LastName LIKE @name
```

```
IF @address IS NOT NULL  
PRINT CONCAT('The address is ', @address)
```

T-SQL – Examples

- Are scripts equivalent?

...

```
SELECT @address = CONCAT_WS(' ',  
                             Country,  
                             City,  
                             Address)
```

```
FROM Employees
```

```
WHERE LastName LIKE @name
```



...

```
SET @address = (SELECT CONCAT_WS(' ',  
                                Country,  
                                City,  
                                Address)
```

```
FROM Employees
```

```
WHERE LastName LIKE @name)
```

Multiple variables can be set

It may not be deterministic
for multiple records - sort

If there is no result,
the value of the variable is not changed

Only one variable can be set

For multiple records,
we receive an error message

If there is no match,
the variable will be NULL

T-SQL – Examples

- Manage multiple results

```
DECLARE @name NVARCHAR(20) = 'King%',
        @address NVARCHAR(max),
        @res_no INT

SELECT @res_no = COUNT(*) -- How many records found?
FROM Employees
WHERE LastName LIKE @name

IF @res_no = 0
    PRINT 'Employee not found'
ELSE IF @res_no > 1
    PRINT 'More than one Employee found'
ELSE BEGIN
    SELECT @address = CONCAT_WS(', ', Country, City, Address)
    FROM Employees
    WHERE LastName LIKE @name
    PRINT 'The address is ' + @address
END
```

T-SQL – Examples

```
DECLARE @name NVARCHAR(20) = 'Fuller%',  
        @address NVARCHAR(max),  
        @res_no INT,  
        @emp_id INT
```

```
SELECT @res_no=count(*)
```

```
FROM Employees
```

```
WHERE LastName LIKE @name
```

```
IF @res_no=0
```

```
PRINT 'Nincs találat.'
```

```
ELSE IF @res_no>1
```

```
PRINT 'Több, mint 1 találat.'
```

```
ELSE BEGIN
```

```
SELECT @address = CONCAT_WS(',', ' ', Country, City, Address), @emp_id=EmployeeID
```

```
FROM Employees
```

```
WHERE LastName LIKE @name
```

```
PRINT CONCAT('Employee ID: ', @emp_id, ', address: ', @address)
```

```
UPDATE Employees
```

```
SET salary = 1.1 * salary
```

```
WHERE EmployeeID = @emp_id
```

```
PRINT 'Egyenleg növelve.'
```

```
END
```

- Other DML instructions can also be used within batches
 - Increase the found employee balance by 10%.

Transactions

Transactions

- Transaction: standalone functional unit

A	C	I	D
Atomicity	Consistency	Isolation	Durability
Transactions are all or nothing	Only valid data is saved	Transactions do not affect each other	Written data will not be lost
Either succeeds completely, or fails completely	Any data written to the database must be valid according to all defined rules	Concurrency control	Completed transactions are recorded in non-volatile memory

Transactions – Atomicity

- All SQL statements can be considered an implicit transaction

```
CREATE TABLE [t1] (  
    [id] INT PRIMARY KEY  
)  
  
CREATE TABLE [t2] (  
    [id] INT PRIMARY KEY,  
    [t1_id] INT REFERENCES [t1]([id])  
)
```

GO

```
INSERT [t1] ([id]) VALUES (1), (3), (4), (5)
```

```
INSERT [t2] ([id], [t1_id]) VALUES (10, 3)
```

GO

```
DELETE [t1]
```

GO

```
SELECT *
```

```
FROM [t1]
```

GO

Can't delete

Implicit transaction

The DELETE statement conflicted
with the REFERENCE constraint ...

All records are exists

A

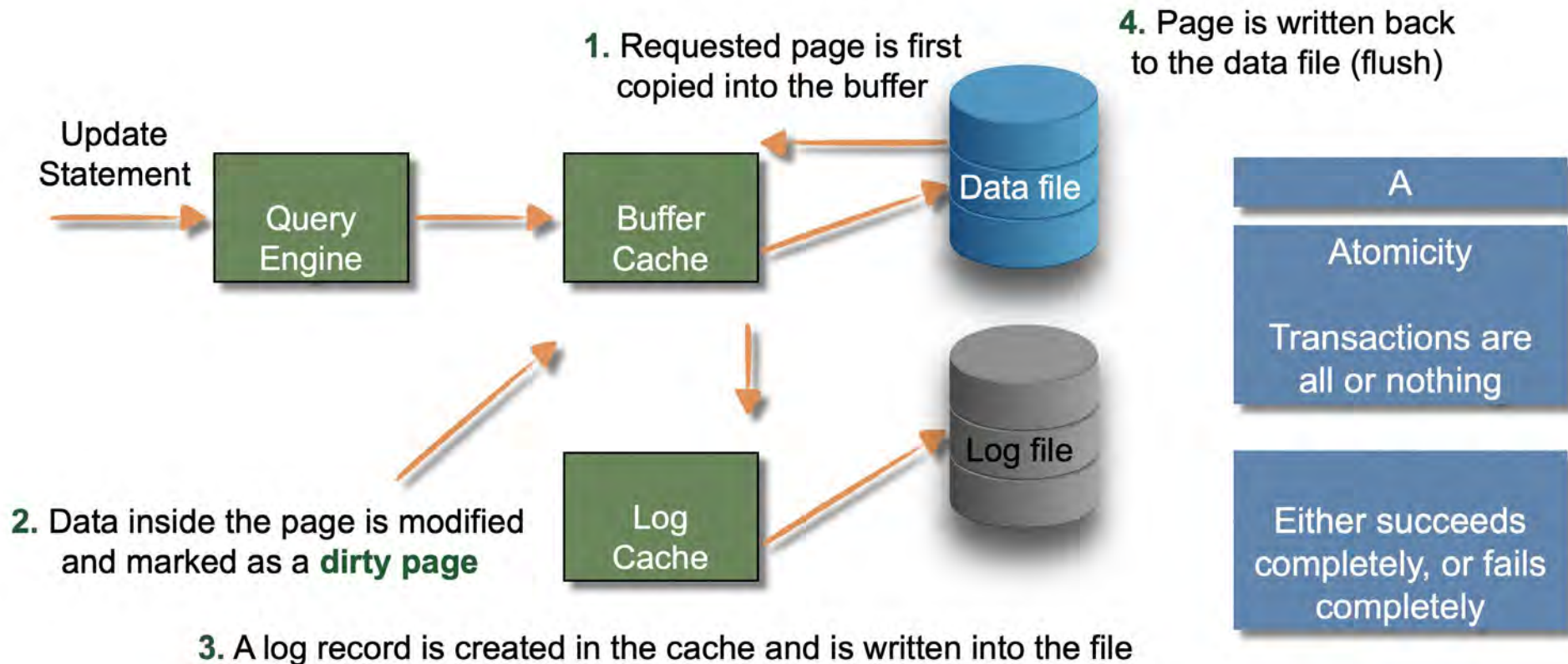
Atomicity

Transactions are
all or nothing

Either succeeds
completely, or fails
completely

Transactions – Atomicity

- It is done through a transaction log



Transaction control

- A database object can be locked
- Properties: owner, object, state, type
- States: wait, grant
- Types: There are many locks, the more important are:
 - S Shared
 - X eXclusive
 - U Update $S \succ U \succ X$
 - I Intent Hierarchical Information

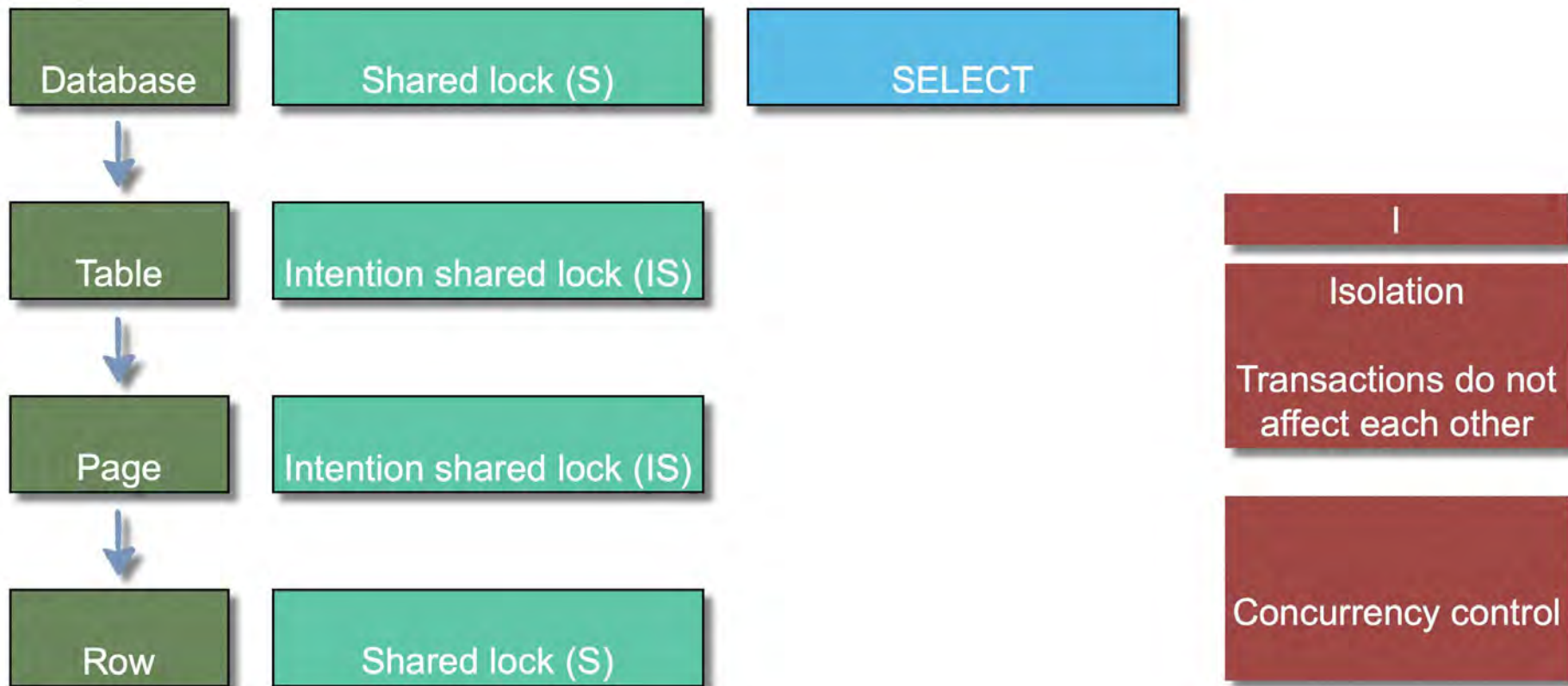
I

Isolation

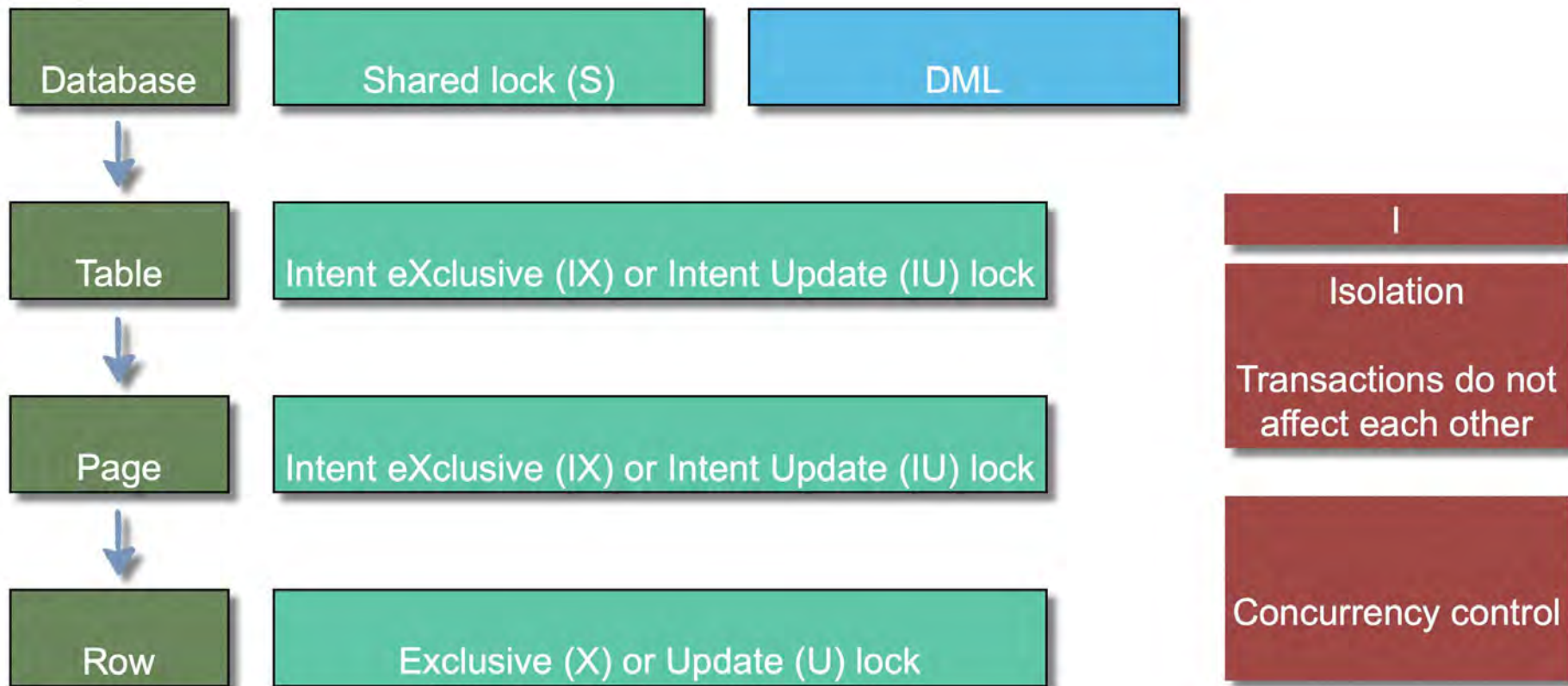
Transactions do not
affect each other

Concurrency control

Transaction control



Transaction control



Transaction control

	Exclusive (X)	Shared (S)	Update (U)
Exclusive (X)	X	X	X
Shared (S)	X	✓	✓
Update (U)	X	✓	X

I

Isolation

Transactions do not
affect each other

Concurrency control

Transaction control – S lock

- **Read uncommitted (dirty read)**
 - The lowest isolation level, dirty reads are allowed
- **Read committed (default)**
 - (S) locks are released as soon as the SELECT operation is performed
 - (U) locks are keep until the end of the transaction
- **Repeatable read**
 - keeps read and write locks that are acquired on selected data until the end of the transaction
 - range-locks are not managed, phantom reads can occur
- **Serializable**
 - The highest level
 - Range-locks are acquired

I

Isolation

Transactions do not
affect each other

Concurrency control

Transaction control

- Set transaction level

```
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED
| READ COMMITTED
| REPEATABLE READ
| SNAPSHOT
| SERIALIZABLE
}
```

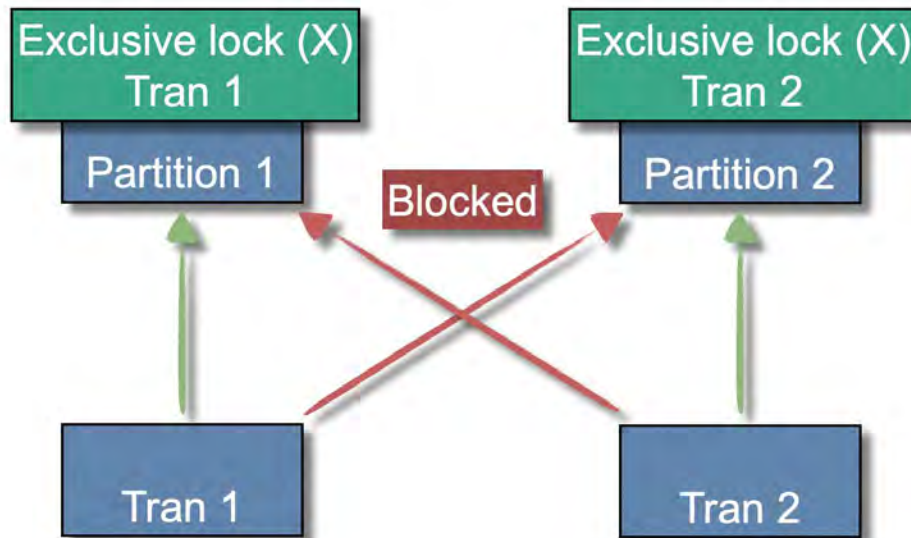
I

Isolation

Transactions do not
affect each other

Concurrency control

Transaction control – Deadlock



I
Isolation
Transactions do not affect each other
Concurrency control

Stored Procedure

T-SQL – Stored Procedures

- **Stored procedures allow you to run T-SQL or CLR (Common Language Runtime) scripts on the server side**
- **The code does not run when it is created, only syntax checking!**
- **The first time you run it, it compiles the server to its native code**
- **Faster execution**

T-SQL – Stored Procedures

- Only need to translate and create an execution plan once (usually)
- Rights can be managed flexibly
- Manage input and output parameters
- All operations that are allowed outside the stored procedure can be performed - even another stored procedure can be called
- There is a status variable for execution state

T-SQL – Stored Procedures

- SQL Server and Azure SQL

```
CREATE [ OR ALTER ] { PROC | PROCEDURE }  
    [schema_name.] procedure_name [ ; number ]  
    [ { @parameter [ type_schema_name. ] data_type }  
        [ VARYING ] [ = default ] [ OUT | OUTPUT | READONLY ]  
    ] [ ,...n ]  
[ WITH <procedure_option> [ ,...n ] ]  
[ FOR REPLICATION ]  
AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }  
[;]
```

```
<procedure_option> ::=  
    [ ENCRYPTION ]  
    [ RECOMPILE ]  
    [ EXECUTE AS Clause ]
```

T-SQL – Stored Procedures

- CLR

```
CREATE [ OR ALTER ] { PROC | PROCEDURE }  
    [schema_name.] procedure_name [ ; number ]  
    [ { @parameter [ type_schema_name. ] data_type }  
        [ = default ] [ OUT | OUTPUT ] [READONLY]  
    ] [ ,...n ]  
[ WITH EXECUTE AS Clause ]  
AS { EXTERNAL NAME assembly_name.class_name.method_name }  
[;]
```


T-SQL – Stored Procedures

- Natively Compiled

```
CREATE [ OR ALTER ] { PROC | PROCEDURE } [schema_name.] procedure_name
    [ { @parameter data_type } [ NULL | NOT NULL ] [ = default ]
      [ OUT | OUTPUT ] [READONLY]
    ] [ ,... n ]
    WITH NATIVE_COMPILATION, SCHEMABINDING [ , EXECUTE AS clause ]
AS
{
    BEGIN ATOMIC WITH (set_option [ ,... n ] )
    sql_statement [;] [ ... n ]
    [ END ]
}
[;]
```

<set_option> ::=

```
    LANGUAGE = [ N ] 'language'
  | TRANSACTION ISOLATION LEVEL = { SNAPSHOT | REPEATABLE READ | SERIALIZABLE }
  | [ DATEFIRST = number ]
  | [ DATEFORMAT = format ]
  | [ DELAYED_DURABILITY = { OFF | ON } ]
```

T-SQL – Stored Procedures

- Azure and Parallel Data Warehouse

```
CREATE { PROC | PROCEDURE } [ schema_name.] procedure_name  
    [ { @parameterdata_type } [ OUT | OUTPUT ] ] [ ,...n ]  
AS { [ BEGIN ] sql_statement [;][ ,...n ] [ END ] }  
[;]
```

T-SQL – Stored Procedures

- Simple examples

```
CREATE PROCEDURE What_DB_is_this  
AS  
SELECT DB_NAME() AS ThisDB;
```

```
EXEC What_DB_is_this;
```

```
CREATE PROCEDURE What_DB_is_that @ID INT  
AS  
SELECT DB_NAME(@ID) AS ThisDB;
```

```
EXEC What_DB_is_that 2;
```

User Defined Function

T-SQL – User Defined Functions

- One return parameter (scalar, table)
- The function cannot modify the database (content, schema)!
- No stored procedure can be called*
- Cannot use try - catch block
- Schema name is required when the function is called (dbo, login name)

T-SQL – User Defined Functions

- Scalar return type

```
CREATE [ OR ALTER ] FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
  [ ,...n ]
]
)
RETURNS return_data_type
[ WITH <function_option> [ ,...n ] ]
[ AS ]
BEGIN
    function_body
    RETURN scalar_expression
END
[ ; ]
```

T-SQL – User Defined Functions

- Scalar return type

```
DROP FUNCTION IF EXISTS dbo.fYearMonth  
GO
```

```
CREATE FUNCTION fYearMonth(@datum DATETIME)  
RETURNS VARCHAR(50) AS  
BEGIN  
    RETURN ISNULL(FORMAT(@datum, 'yyyy_MM'), 'N.A')  
END  
GO
```

```
SELECT [e].[EmployeeID], [LastName], [dbo].fYearMonth([OrderDate]) AS [Month],  
       COUNT([OrderID]) AS [OrderCount]  
FROM [Employees] [e]  
LEFT OUTER JOIN [Orders] [o] ON [e].[EmployeeID] = [o].[EmployeeID]  
GROUP BY [e].[EmployeeID], [LastName], [dbo].fYearMonth([OrderDate])  
ORDER BY [LastName], [Month]
```

T-SQL – User Defined Functions

- Table return type (inline)

```
CREATE [ OR ALTER ] FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
  [ ,...n ]
]
)
RETURNS TABLE
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    RETURN [ ( ) select_stmt [ ) ]
[ ; ]
```

T-SQL – User Defined Functions

- Table return type (inline)

```
DROP FUNCTION IF EXISTS dbo.fProductsOfCategory
```

```
GO
```

```
CREATE FUNCTION fProductsOfCategory(@product_id INT)
```

```
RETURNS TABLE AS
```

```
RETURN (
```

```
    SELECT p.[ProductName], COUNT(*) AS [OrderCount]
```

```
    FROM [Products] p
```

```
    JOIN [Order Details] od ON p.[ProductID] = od.[ProductID]
```

```
    WHERE p.[CategoryID] = @product_id
```

```
    GROUP BY p.[ProductID], p.[ProductName]
```

```
)
```

```
GO
```

```
SELECT *
```

```
FROM dbo.fProductsOfCategory(2)
```

```
WHERE [OrderCount] > 30
```

T-SQL – User Defined Functions

- Table return type

```
CREATE [ OR ALTER ] FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
  [ ,...n ]
]
)
RETURNS @return_variable TABLE <table_type_definition>
[ WITH <function_option> [ ,...n ] ]
[ AS ]
BEGIN
    function_body
    RETURN
END
[ ; ]
```


T-SQL – Trigger

- **A stored procedure that is bind to an activities - it runs automatically**
- **DML**
 - it always runs, even if the number of records involved is zero
 - Run instead of an operation or after an operation
- **DDL**
 - Fired by CREATE, ALTER or DROP
- **Logon**
 - Fired by user login
- **This can call a stored procedure, function, use T-SQL statements**
- **This can trigger to run an another trigger (Nested)**

Triggers

T-SQL – DML Trigger

```
CREATE [ OR ALTER ] TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { sql_statement [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier [ ; ] > }
```

```
<dml_trigger_option> ::=
    [ ENCRYPTION ]
    [ EXECUTE AS Clause ]
```

```
<method_specifier> ::=
    assembly_name.class_name.method_name
```

T-SQL – DML Trigger

- There are no input parameters and return values
- Trigger failure -> Rollback
- Two virtual tables are available: **inserted** and **deleted**
- The structure of the basic and the virtual tables are same!

table/operation	inserted	deleted
insert	inserted records	empty
update	modified records	original records
delete	empty	deleted records