KUL

Principles of Database Management

PART I : Prof Dr. Bart Baesens

Ysaline de Wouters

2015 - 2016

CHAPTER 1 : FUNDAMENTAL CONCEPTS

1. Introduction

Nowadays, data is everywhere and in many different forms and volumes. Databases and database technology have a major impact on the growing use of computers. All this data must be stored and managed.

Data : facts that can be recorded and that have implicit meaning. Ex : names, telephone numbers, addresses, … Therefore, a **database** is a collection of related data within a specific business process or problem setting. A database has a target group of users and applications. Databases can be of any size and complexity. A famous example of a large commercial database is Amazon.com.

There are many traditional database applications:

- Storage and retrieval of traditional numeric and **alphanumeric data**. Ex : Stock administration : keeping track of the number of products in stock
- Storage and retrieval of **multimedia data**: pictures, audio, video. Ex : YouTube, spotifiy
- Storage and retrieval of **biometric data**. Ex: fingerprint, retina scan, wearables, …
- Storage and retrieval of **geographical data** : store and analyze maps, wheater data and satellite image
- Storage and retrieval of extremely **volatile data**: high frequency trading, sensors, …
- Storage and retrieval of **Web content**. Ex : HTML, XML, PDF, … Twitter, Facebook, Google
- Storage of large datasets for **analytics**. Ex: Walmart

A database management system, referred as **DBMS**, is a collection of programs that enables users to create and maintain a database. In other words, DBMS is a software package which allows defining, store, using and maintaining a database. A DBMS consists of many modules with specific functionalities. Ex : Oracle, Microsoft, IBM

- A **database system** consists of the combination of a DBMS and a database

2. Characteristics of the Database Approach

A number of characteristics distinguish the database approach from the much older approach of programming with files.

- **Waste of storage space because of duplicates** (= redundant data). In traditional file processing, each user defines and implements the files needed for a specific software application as part of programming the application. Ex: one user, the grade reporting office, may keep files on students and their grades. Programs to print a student's transcript to enter new grades are implemented as part of the application. A second user, the accounting office, may keep track of students' fees and their payments. Although both users are interested in date about students, each user maintains separate files.

- In the database approach, a single repository maintains data that is defined once and the accessed by various users. The names or labels of data are defined once, and used repeatedly by queries, transactions and applications, …
- **Inconsistent data**. Ex: customer data changed in only one file.
- **Strong dependency** between applications and data. That is to say that, change in data file necessitates changes in all applications that use the data. DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS cataloc separately from the access programs. **= program-data independence**
- Concurrent actions can lead to **inconsistent state of date**. A database typically has many users, each of whom may require a different perspective or view of the database. A view may be a subset of the database or it may contain virtual data that is derived from the database files but is not explicitly stored. For instance, application 1 performs cash transfer while application 2 calculates account balance.
- **Difficult to integrate various applications**. A multiuser DBMS must allow multiple users to access the database at the same time. It must also include concurrency control software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct. Ex: when several reservation agents try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one agent at a time for assignment to a passenger.

3. Elements of a Database System

3.1 Data model

A data model is a clear description of the data concepts, their relationships and various data constraints that together make up the content of the database. There exist different types of data models.

- **Conceptual data model:** This type is used as a communication tool to make sure that data requirements are adequately capped and modeled. It should be implementation independent, user friendly and close to how business users perceive the data.
  Ex **:** (E)ER, object-oriented models
- **Logical data model :** it is a kind of translation or mapping of the conceptual Data models. The concepts may be understood by business users but are not too far removed from physical data organization.
  Ex : Hierarchical model, network model, relational model, object oriented model.
- **Physical data model :** this low level concept clearly describes which data will be stored where, which format, …

3.2 Schemas and instances

In any data model it is important to distinguish between the description of the data and the data itself.

On the one hand, there is a **database schema** which is stored in the catalog. This schema is a description of a database, which is specified during database design. It is not expected to

change too frequently. On the other hand there is the **database state** which corresponds to the data in the database at a particular moment, also called the current set of instances. It changes on ongoing basis.

STUDENT

Number	Name	Address	Email
0165854	Bart Baesens	1040 Market Street, SF	Bart.Baesens@kuleuven.be
0168975	Seppe vanden Broucke	520, Fifth Avenue, NY	Seppe.vandenbroucke@kuleuven.be
0157895	Wilfried Lemahieu	644, Wacker Drive, Chicago	Wilfried.Lemahieu@kuleuven.be

Database schema

STUDENT (number, name, address, email)

COURSE (number, name)

BUILDING (number, address)

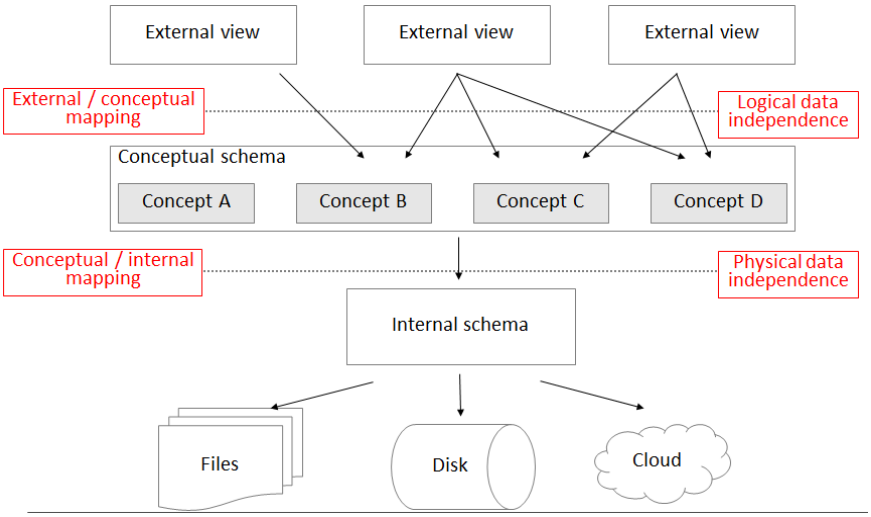
COURSE

Number	Name
D0I69A	Principles of Database Management
D0R04A	Basic Programming
D0T21A	Big Data & Analytics

BUILDING

Number	Address
0589	Naamsestraat 69, Leuven
0365	Naamsestraat 78, Leuven
0589	Tiensestraat 115, Leuven

3.3 The three-schema architecture



The three-schema architecture is an essential element of any database environment or application. This figure illustrates this architecture.

Let's start with the **middle layer** : here we have the conceptual data scheme. As mentioned before, it focuses on data concepts, their relationships and data constraints without bothering about the actual DBMS implementation. It should be an user friendly and transparent data model. It would be matched to a logical data model based on a close collaboration between DB designer and users. The middle layer includes both conceptual as logical scheme. The **top level** represents the external scheme. Here, views can be defined which offers a window on a carefully selected part of conceptual model. The external views will be tailored to what's a data need of application. A view conserves one or more applications. Ex : view offering only student information or only building information to capacity planning application.

- **External view**: Each external schema describes the part of the database that a particular user group is interested in and hides the rest of database from that user group. It allows controlling the data access.
- **Conceptual scheme**: Specifies data concepts, characteristics, relationships, integrity rules and behavior.

- **Internal scheme:** specifies how data is stored or organized physically. Ex: indexes, access paths, etc.

- Ideally, changes shouldn't have impacts on the other parts or at least, changes in one layer should have minimal impact on the other layers.
  This three-schema architecture has **advantages** in terms of efficiency, maintainability, security and performance.

3.4 Data dictionary (catalog)

The data dictionary or catalog constitutes the heart of the database system. It contains the data definition or Metadata of database application. More specifically it stores the definitions of the external, conceptual and physical scheme. It also synchronizes these three schemes to make sure consistency is guaranteed.

3.5 DBMS languages

- Every DBMS comes with one or more data base languages.
  - **Data Definition Language (DDL**) : This language is used by the database administrator to define the database's logical, internal and external schemas. It is stored in the catalog.
  - **Data Manipulation Language (DML) :** This language is used to retrieve, insert, delete and modify data. DML statements can be embedded in a general-purpose programming language or entered interactively through a front-end querying tool.
- For relational database systems, SQL is both the DDL and DML. It can be used interactively *(= interactive SQL)* or embedded in a programming language (*= embedded SQL).*

3.6 Database users

For a small personal database, such as the list of addresses, one person typically defines, constructs, and manipulates the database, and there is no sharing. However, in large organizations, many people are involved in the design, use and maintenance of a large database with hundreds of users.

- **Database Designer:** He is responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. Therefore he is responsible for designing the conceptual schema. These tasks are mostly undertaken before the database is actually implemented and populated with data. He closely interacts with all prospective database users to make sure to understand their requirements and to create a design that meets these requirements.
- **Database Administrator:** He is responsible for administering the resources. The DBA is responsible for authorizing access to the database, coordinating and monitoring its use and acquiring software and hardware resources as needed. Moreover, he designs the external and physical schema. He also set up the database infrastructure

and monitors its performance by inspecting KPI indicators (e.g. response time and storage consumed).

- **Application developer:** Responsible for developing the database applications in a programming language such as Java. He will provide the data requirements which will then be translated by the DBA to few definitions.
- **Business users :** Will run the applications to perform specific data operations. He can also query database using interactive facilities.
- **DBMS vendors :** offers the DBMS software packages. Ex: Oracle, Microsoft and IBM.

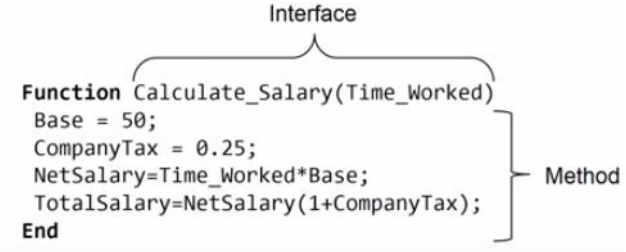
4. Advantages of using database design

4.1 Data and functional independence

Data independence refers to the fact that changes in data definitions have minimal to no impact on the applications using the data. These changes may occur on both the physical or logical level.

- **Physical data independence** implies that neither applications nor the external or conceptual scheme must be changed when changes are made to the data storage specifications. Ex : reorganizing data across different storage facilities. The app will keep on running successfully maybe even faster because of **reorganization of data.**
  Nonetheless, it's important that DBMSs provides interfaces between conceptual and physical data models.
- **Logical data independence** implies that software applications are minimally impacted by changes in the conceptual schema.
  Ex : adding new data concepts, characteristics or relationships, …
  DBMS provides interfaces between the conceptual and external schema to guarantee success and data independence.

DBMS may also allow storing functionalities such as small routines or functions. Every function has an interface or signature containing its name together with its input and output. The method or implementation specifies how the functions should be executed and is also stored in the database. **Functional independence** refers to the fact that implementation can change without having any impact on software applications.



This is an example of a function. Its name is "calculate_salary". It has one input : Time worked which reprensents the number of hours worked by an employee. The method then calculates the total salary taking into account the base salary per hour and the company tax.

Functionally independent means that the method may change without having any impact on the application calling the function.

4.2 Database modeling

Another key advantage is the construction of a data model. The data model is a representation of the data concepts together with characteristics and relationships. It can also contain integrity rules and functions. It should provide a formal and perfect mapping of the real world within a close collaboration with the business user. But the case with perfect mapping is often unrealistic. It is important that the date model assumptions and shortcomings are clearly documented.

Popular examples of the data models are: hierarchical model, CODASYL model, (E)ER model, relational model and the object oriented model.

4.3 Managing data redundancy

In the Database approach, redundant data can be successfully managed. DBMS will be responsible to manage the redundancy by providing synchronization facilities to guarantee the data consistency. Ex: update of local data copy will be automatically propagated to all other data copies. DBMS also guarantees the correctness of data. It also requires no user intervension.

4.4 Specifying integrity rules

  - **Syntactical rules** specifies how data should be represented and stored. Ex : numeric customerID; birthdate as DD/MM/YYYY.
  - **Semantical rules :** It focusses on the correctness and meaning of data. Ex : unique customerID should be unique ; account balance >0; customer with pending invoices cannot be deleted.
- In the file approach, these integrity rules had to be embedded in the applications. In the database approach they are specified as part of the conceptual scheme and stored centrally in the catalog.

  Improves efficiency and maintanance of the applications. Integrity ryles are enforced by the DBMS whenever anything is updated (date loading, data manipulation, ...) in the database approach. They are also enforced by the applications accessing the files in the file based approach resulting in dupplication of codes with risks of inconsitencies.

4.5 Concurrency control

DBMS most ensure the **ACID** properties :

- **Atomicity** : all-or nothing property
- **Consistency** : a transaction brings the database form one consistent state to another
- **Isolation** : concurrent transactions appear to run in isolation
- **Durability** : DBMS ensures durability of transactions so that they can be made permanent.

4.6 Data security

Some users have a read access while others have a write access to data. This access can also be defined to certain points of the data which means some users have access to the whole database while others only to certain parts.

Trends such as e-business (B2B, B2C), CRM, … stress the importance of data security. Data access can be managed via logins and passwords assigned to users or user accounts. Each account has its own authorization rules, which are stored in the catalog.

4.7 Backup and recovery facilities

DBMS allows to availibilty of backup and recovery facilities in case of loss of data which might be due to hardware or network errors or bugs in the system or application software for example.

- **Backup facilities** perform full or incremental backups.
- **Recovery facilities** allow to restore data after loss or damage occurred.

4.8 Performance utilities

Performance utilities provide specific functionalities. Here are some examples.

- Distributing data storage;
- Tuning indices to allow faster queries;
- Tuning queries to improve application performance;
- Optimizing buffer management.

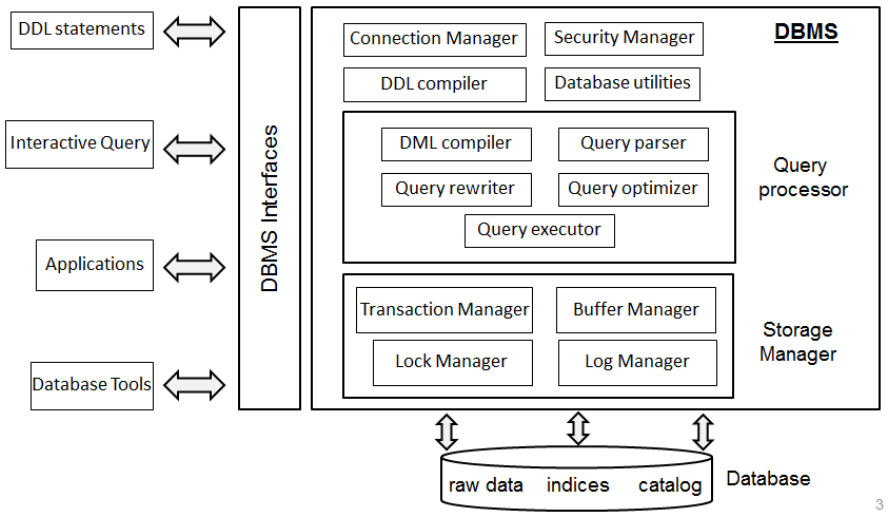
This is part of the job of the DBA.

CHAPTER 2 : ARCHITECTURE AND CLASSIFICATION OF DBMSs

The architecture of DBMS database management system packages has evolved from the early monolithic systems, where the whole DBMS software package was one tightly integrated system, to the modern DBMS packages that are modular in design, with client/server system architecture.

1. Architecture of a DBMS

A DBMS is a complex software system. This section is meant to discuss the types of software components that constitute a DBMS and the types of computer system software with which the DBMS interacts.



This is an overview of the key components of DBMS. This figure is by no means exhaustive! Some components may be left out, some others added.
To the left there are various ways to interact with DBMS. DDL statements are meant to create data definitions which will be stored in a catalog. Interactive queries are typically executed from a front-end tool such as a comment line interface. Applications interact with DBMS using embedded DML statements. The DBA can use various database tools to maintain the DBMS.
To facilitate the usage of this, the DBMS will provide various interfaces which will invoke the components. The most important components are : the connection Manager, the security manager, the DDL compiler, the query processor and the storage manager.

1.1 The Query Processor

The query processor is one of the most important parts of the DBMS. It will assist in the execution of database queries such as retrieval of data, insertion, update and removal of data from the database. The query processor consists of :

- **DML compiler** : it provides a set of constructs to select, insert update and delete data.

DML :

- **Procedural DML :** specifies how to navigate in the database to locate and modify the data. Usually starts by positioning on one specific record and navigate from there onwards to other records using memory pointers. Procedural DML is also referred to as **record at a time** DML. DBMSs with procedural DML do not have a query processor available. In other words, the application developer has to explicitly define the optimization.
  ➔ This isn't the preferred implementation since it complicates the efficiency, maintenance and transparency of database applications.
- **Declarative DML** : More efficient implementation. The DML statements specify what data should be retrieved or what changes should be made rather than how they should be done. The DBMS will then determine the physical execution in terms of access for a navigational strategy. It is usually **set-at-a-time DML.**
  Ex: SQL

Applications often need to access the database. DML is embedded in the application in order to access the database. The application is written in the host language (Ex : Java). Some DML code is embedded inside the application (Ex : embedded SQL).
Data structures of DBMS and DML may be different from the data structure of the host language. This is often referred to an Impedance Mismatch problem. This problem can be solved in various ways.

- Choose a host language and DBMS with comparable data structures. EX combine java with object oriented DBMS
- Opt to use middleware to map data structures from the DBMS to the host language and vice-versa.

The DML compiler will start by extracting the DML statements from the host language. It will collaborate with the query parser, query rewriter, query optimizer, query executor for executing the DML statements. Errors will be generated and reported if necessary.

- **Query parser** will parse the query into the internal representation format. Il will check the query for syntactical and semantically correctness. To do so, it will make extensive use of the catalog to verify if the integrity constraints have been respected. Again errors will be generated and recorded if necessary.
- **Query rewriter** will optimize the query independent of the current database state. It will simplify it using predefined rules and heuristics. Nested queries might be reformulated or flattens to join queries.
- **Query optimizer** he will optimize the query based on the current database state. Can make use of pre-defined indices which are part of the physical scheme and provide a quick access to the date. He will come up with various execution plans and evaluate their costs by aggregating the estimate number of input/output operations, …
- **Query executer**

1.2 The Storage manager

Storage manager includes :

- **Transaction Manager** supervises the execution of database transactions. A database transaction is a sequence of retried operations considered to be an atomic unites. This manager will create a schedule to improve efficiency and execution performance. It also guarantees the ACID properties in a multiuser environment.
- **Buffer Manager** is responsible for managing the buffer or cache of the DBMS for speedy access. He is responsible for intelligently cashing the date in the buffer for speed access. It needs to constantly monitor the buffer to decide which content should be removed and which one added. In case data in the buffer has been updated, it must also synchronize the corresponding external memory. It must also be able to serve multiple transactions simultaneously.
- **Lock manager** essential for providing concurrency control. Before a transaction can read or write a database object, it must acquire a lock. A read lock allows a transaction to read a database object whilst a write lock allows a transaction to update. To enforce a transaction atomicity consistency, a lock database object may prevent other transactions from using it.
  - ➔ Locking protocol which describes the locking rules and locking tables with the lock information
- **Log Manager** keeps track of all database operations in a log file. He will be called upon to undo actions of aborted transactions or during crash recoveries.
  All these components interact in various ways depending on which interface part is being executed.
  The database itself contains the raw data, the indices and the catalog.

1.3 DBMS

- **The connection manager** allows setting up a database connection. It can be set up locally as well as through a network. He will verify the authentication information such as username and password and will return in connection handle. The database connection can either run as a single process or as a thread within a process. A **thread** represents an execution plot within a process. It represents the smallest unit of processor scheduling. Multiple threads can run within a process and share resources.
- The **security manager** will verify whether a user has the right privileges to execute the database actions required. Some users can have read access whilst others have write access. This can be refined to certain parts of the data. He will retrieve these privileges from the catalog.
- **The DDL compiler** compiles the data definitions specified in DDL. Ideally the DBMS should foresee 3 DDLS : one for the physical scheme, one for the conceptual scheme, one for the external scheme. Often there is just 1 DDL with 3 independent sets of instructions. This is the case for most relation databases which use SQL as their DDL. The DDL compiler will first parse the DDL statements and check there correctness and then translate it to an internal format and generate errors if required. Then he will register the data definitions in the catalog where it can be used by all the other components of the DBMS.

1.4 Database utilities

- **Loading utility** allows providing the database with information from a variety of different sources such as another DBMS text file, shell files, etc.
- **Reorganization utility** automatically reorganizes the data in order to improve the performance. It can also create new access paths to improve the performance.
- **Performance monitoring** utilities report various KPI's such as storage space consumed. It also provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files or whether to add or drop indexes to improve performance.
- **User management utilities** allow creating user groups or accounts and assigning privileges to them.
- **Back-up and recovery utility :** creates a backup copy of the database, usually by dumping the entire database onto tape or other mass storage medium. The backup copy can be used to restore the database in case of catastrophic disk failure. **Incremental backups** are also often used, where only changes since the previous backup are recorded.

- Other utilities may be available for sorting files, handling data compression, monitoring access by users, interfacing with the network, and performing other functions.

1.5 DBMS interfaces

A DBMS needs to interact with many different parties such as a database designer, a database administrator and application or even the end user. To facilitate the communication, it will provide many different interfaces.

- Web-based interfaces
- Stand-alone query language interfaces
- Command line interface
- Forms-based interface
- Graphical user interface
- Natural Language interfaces
- Admin Interfaces for the DBA
- Network interfaces

2. Classification of DBMS

Can be that the DBMS falls in many categories and also other categories can be considered.

2.1 DBMS based on data model

- Various types of data models have been introduced.

- **Hierarchical DBMS** adopt the hierarchical data model. In this case, the DML is procedural and record oriented which means there is no query processor included. The definitions of conceptual and physical schema are intertwined. Popular examples are IMS and Registry in Microsoft Windows.
- **Network DBMSs** use a network data model. One of the most popular types are CODASYL DBMSs. Also here the DML is procedural and record oriented and there is no query processor. Consequently, the definitions of the conceptual and physical scheme are also intertwined. Popular examples are CA-IDMS, UDS, DMS 1100, IMAGE, …
- **Relational DBMSs** use the relation data model and are the most popular in industry nowadays. They use SQL who structured query language for both DDL and DML operations. SQL is declarative and set oriented. The query processor is provided to optimize and execute the database queries. Data independence is available thanks to a strict separation between the conceptual and physical scheme. This makes it very attractive to develop powerful database applications.
  Ex : MySQL, DB2, …
- **Object-oriented DBMS** : object-oriented data model. It's about objects with variables and methods. There is no impedance mismatch. These kinds of models are not that popular in the industry due to their complexity. Ex : db4o, ObjectDB, Gemstone.
- **Object-relational DBMS** or extended relational data models uses relational model extended with OO concepts. Ex : User-defined types, functions, collections, inheritance, behavior. The DML is SQL which is declarative and set oriented.
  Ex : Oracle Data base, DB2, Microsoft SQL Server.

2.2 DBMS based on simultaneous access

DBMSs can also be categorized base on the degree of simultaneous access.

- In a single-user system, only one user at a time is allowed to work with the DBMS. Obviously this is not desirable in a network environment.

- Multi-user systems allow multiple users to simultaneously interact with the database in a distributed environment. To do so successfully, the DBMS should support multi-threading and provide facilities for concurrency control. A dispatcher control will then distribute the various incoming database requests among server instances threats.

2.3 DBMS based on architecture

- In a centralized DBMS architecture, the data is maintained on a centralized server. All queries will then have to be processed by the single server.
- In a client-server DBMS, active clients will request services from passive servers. A flat client stores more processing functionality on the client's.
- The n-tier DBMS architecture is a straightforward extension of the client-server architecture. Middleware for transparent communication.
  Ex: Client with the GUI functionality
- Cloud DBMS architecture: the DBMS and database will be hosted by a third party cloud provider. The data can then be distributed across multiple computers in a network such as

the internet. Although this is a cheap solution, it is often less performing. Ex: Apache Cassandra project and Google's big tables.

- Federated DBMS is a DBMS which provides a uniform interface to multiple underlying date sources such other DBMS, file system, document management system. It aims at highlighting the underlying storage details to facilitate data access.
- In-memory DBMS stores all data in internal memory instead of external expensive storage such as a disk. It is used for real-time applications. Periodic snapshots to external storage can be taken for data persistence. Ex : SAP Hanta

2.4 DBMS based on usage

DBMSs can also be characterized based on usage :

- Operational versus tactical/strategic usage
    - **OLTP DBMS** stands for Online Transactions Processing. It focuses on managing operational data.
      Ex : a point of sale (POS application) in a supermarket where date needs to be stored about each purchase transactions such as customer information, products purchased, prices paid, timing, … For each application, the database server must be able to process lots of simple transactions per unit of time. Also transactions are initiated in real time simultaneously by lots of users and applications. Hence DBMS must of good support for processing high volumes of short, simple queries.
    - OLAP DBMS stands for Online Analytical processing focuses on using operational data for tactical or strategically decision making. Here, a limited number of users will formulate complex queries to analyze huge amounts of data.
- **Big data & Analytics:** they are all around these days. New database technologies have been introduced to efficiently cope with Big data. NoSQL stands for not only SQL and is one of these new technologies. NoSQL databases abandon the well-known and popular relational scheme in favor of a more flexible, schema less database structure. This is handy to store unstructured information such as emails, tax documents, Twitter tweets, Facebook posts, etc.
  One of the advantages is that they more easily horizontally scale in terms of storage. There exist 4 popular types of NoSQL database technologies.
- **XML DBMS** : Database representation used to represent the data.
    - **Native XML DBMS** : Stores XML data bay using the logical intrinsic structure of the XML document. They map the hierarchical or three structure of an XML document to a physical storage structure.
    - **Enable XML DBMS** : Existing DBMS which are extended with facilities to store XML data and structured data in an integrated and transparent way.
      ➔ Both types of XML DBMSs also provide facilities to query XML data.
- Multimedia DBMS allow the storage of multimedia data such as text, images, audio, video, 3D games, CAD designs, etc. They should also provide content based query facilities. Ex : "Find images of bart". Streaming facilities should also be included to stream multimedia

output. These are resource intensive transactions. A multimedia data is usually stored as a binary large object (BLOB).

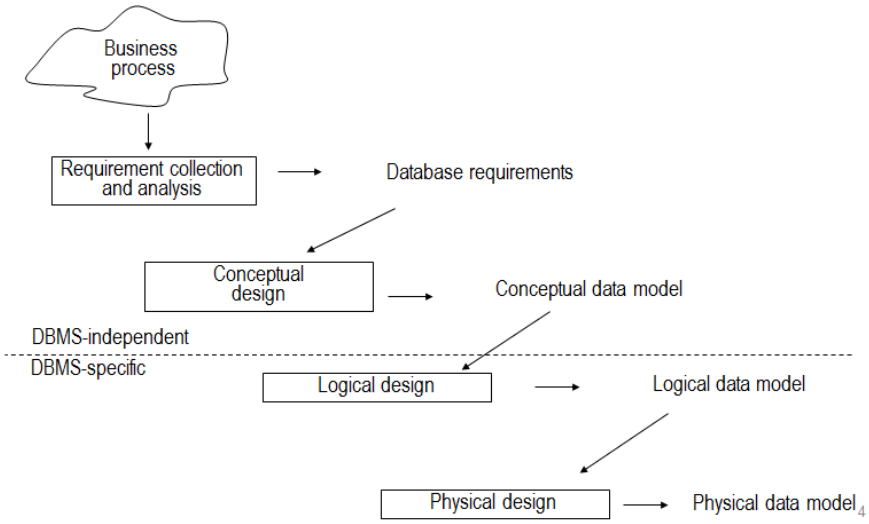
- **GIS DBMS** allows to store and query spatial data.
- **Sensor DBMS** allows managing sensor data such as biometric data or telematics data. It should allow formulating applications specific queries.
- **Mobile DBMS** are the type of DBMSs running on smartphones, tablets and other types of devices. They should obviously have special properties. They should always be online, have a small-footprint with a limited processing power, storage and battery life. They could connect and synchronize to a central DBMS. Ideally they should be able to handle local queries and also support self-management without the intervention of a DBA. Ex : Oracle Lite, Sybase SQL, …
- **Open source DBMS** are DBMSs of which the code is publically available and can be extend by anyone. Hence, this has the advantage of having a large development community working on a project. They are very popular for small business applications and in developing countries where budgets are limited. They can be obtained from a well-known website. Ex : MySQL, Postgres SQL, …

CHAPTER 3 : DATA MODELS

The models can be used for either conceptual modeling or physical modeling or logical modeling.

1. Concepts of data modeling

Developing a **conceptual scheme** or model is the **first step** of a database design. The goal is to adequately capture the specifications of the data and constraints to be represented in the database. To do so, the **database designer** and the **end user** should closely collaborate. The main focus is to capture the semantics of the business process as accurately as possible without being concerned about DBMS-related issues. This purpose requires a high-level model, such as the (E)ER model. This allows the database designer to capture data requirement end provide a user friendly representation to the end user so that he or she can also accurately understand what has been modeled. If the model is agreed by both parties, we can proceed to the database design.



These are the various steps of the database design. We start from a business process and think about the procurement application, invoice handling process, salary administration or logistics process. Database designer and end user will closeley collaborate to write down the **data requirements** of the process in a conceptual data model. During thhis step, the database designers interview prospective database users to understand and document their data requirements. The result of this step is a concesely written set of users'r requirements. As mentioned before, there should be a hihg-level model that is easy to understand from the business user and fromal enough to the database designer. This conceptual model will have its limitations …

Once the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This is the so called **conceptual design**.

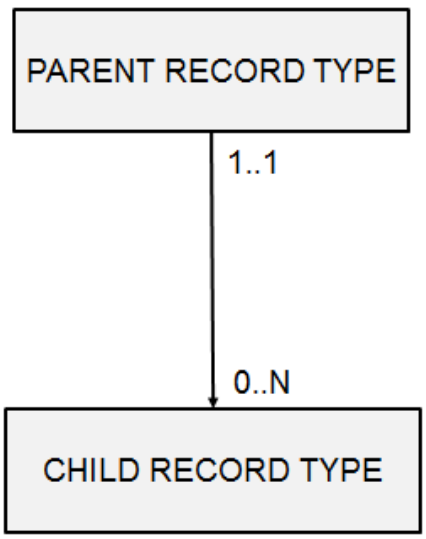
Once all parties have agreed, it can be mapped to a **logical data model** which take into account the implementation environment. That is to say that the logical data model is the data model that will be used by the DBMS for actual implementation.
In a final step, the logical data model can be mapped to a **physical data model**. During this mapping process, semantics could get lost or added. During this desing, the internal storage strucutres, file organisations, indexes, access paths, … are sepcified. In a parallel with these activities, application programs are designed and implemented as database transactions corresponding to heigh-level transaction specifications.

2. Hierarchical model

The hierarchical model is one of the first data models that was developed. It was developed during the Apollo project. It is purely based on a hierarchical data model. No formal description is available and it has several structural limitations and is considered to be legacy.
The two key building blocks are:

- **A record type** is a set of records describing a set of similar entities. Ex: a product record type or a supplier record type. It has **0, 1 or more records**. It consists out of data fields or data items. Ex: a product record type has a field product name, product number, product color, etc.
- **Relational type** connects two records type. It models the relationships between records type. Only *1 to n* relationship types can be modeled. Hence, a parent record type can have multiple child record types. But a child record type has at most one parent record type. Relationship types can be nested. Hence the child record type can be parent in another relationship type, which allows building hierarchical structures. The **root record** type is the one the sits at the top of the hierarchy. A **leaf-record** type sits at the bottom of the hierarchy.

We have a parent and a child record type. The parent record can be connected to minimum zero max N child records whereas the child can only be connected to minimum one or maximum one parent record type. Hence, the child record is always connected to exactly one parent. No other cardinalities are supported in the hierarchical models. Therefore, it makes it very restrictive in terms of expressive power. The relationship type is always summarized in terms of the maximum cardinalities. Here we can say the hierarchical model only supports 1 to N relationship types.



In this model, all record data needs to be retrieved by navigating from the root node of the hierarchical structure. In other words, the DML adopted is procedural, which is not so nice. It is also limited in terms of expressive power.

Besides, there is no straightforward way to model N:M and 1:1 relationship types. To implement a N:M relationship type, we can assign one record type as the parent and the other as the child record type. We transform the network structure to the three-structure. Every relationship attribute can be put in a child record type. But this solution will create redundancies.
Another alternative is to create two hierarchical structures and connect them using a virtual child-record type and a virtual parent/child relationship type. Pointers can then be used to navigate between both structures. The relationship type attributes can be put into a virtual child record type. This solution has no more redundancies since multiple virtual Childs can refer to one virtual parent. This structure has no more redundancies, it's clear but not so nice to maintain.

1:1 relationship types are a special case of a 1 to N relationship type with N = 1. This relationship cannot be supported in the hierarchical model. The application programs should take care of this !

The hierarchical models only allow relationships types of degree 2, in other words, with two participating record types. Recursive relationship types, with degree 1, need to be implemented using virtual child record types. The maximum and minimal cardinality is one. A child cannot be disconnected from its parents. This implies that once a parent record is removed then all connected child records will be removed as well.

3. CODASYL model

This model was developed by the Data Base Task Group of the Conference on Data System Languages.
It has various popular software implementations. Ex : IDMS, CA-IDMS. It is an **implementation of the network model** which originally includes record types and links and allow a 1:1, 1:n and N:M relationship types. This model only includes record types, set types and 1:N relationship types.

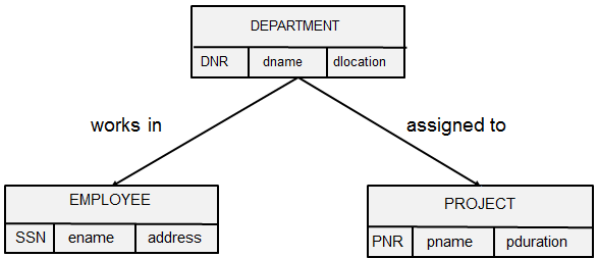
The CODASYL model is considered to be legacy and has a lot of structural limitations.
The **two key building blocks** are record types and set types.

- As in the hierarchical model, **the record type** is a set of records describing a similar entity. It has 0, 1 or more records or record occurrences. It consists out of various data models. Ex: supplier can have various record types such as supplier name, supplier number, …
  The CODASYL model provides support for vectors and repeated groups.
  - A **vector** is a multivalued record type or an atomic data item for which a record can have multiple values. Ex : if a supplier can have multiple email addresses, it could be modeled using a vector.
  - A **repeated group** is a composite data item for which a record can have multiple values. Ex: if a supplier can have multiple addresses each with a zip code, street name, and city, then this can be modeled using a repeated group.
- A set type models a 1:N relationship type between an owner record type and a member record type. A set type has a set occurrence for each record occurrence of the owner record type. A set occurrence has one owner record and 0,1 or more member records. The CODASYL interpretation of a set does not perfectly correspond to the notion of a set used in

mathematics. In mathematics a set is a collection with similar elements and without ordering. A CODASYL set has both an owner and a member record and it's also **possible to order the records.**

- **Differences with the hierarchical model**
  - Support for vectors and repeated groups
  - Members can be disconnected from their owner since the minimal cardinality is equal to zero.
  - Record type can be member record type in multiple set types. Allow the creation of network structures (simple networks).
  - Multiple set types are possible between the same record types.

These models are usually represented using a **network or diagram**.

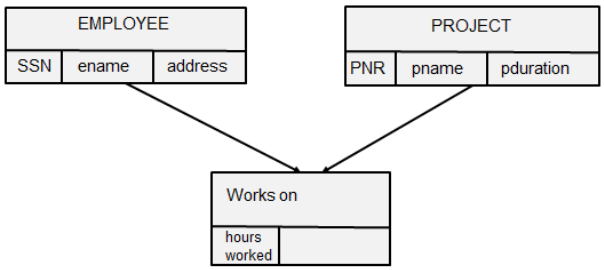


3 record types: Department, employee and project. The department has 0:N employees. Employee works with minimum zero, maximum 1 department. A department can work on 0:N projects. A project is assigned to minimum zero maximum one department.

Again, **1:1** relationship types have to be enforced/modeled in application programs. N:M relationship types needs to be modeled using a workaround.

- A member-record can only belong to one occurrence of a specific set; hence it cannot just define a set type between the corresponding record types.
  One option is to introduce a **dummy record type** which is included as a member record type in two set types having as owners the record types of the original N:M relationship type.
  This dummy record type can also contain the attributes of the relationship type.

N:N relationship type between employee and project. An employee can work to 0 to N projects and a project can be worked upon by 0 to N employees. We modeled this in CODASYL by using a dummy record type "works on" which also includes the attribute type of the relationship "hours worked" representing the number of hours an employee works on a project.



This relationship types has serious **implications** for data usage. Suppose we have a query that asks for all projects an employee is working on. To solve this query, we first need to select a set in a set type employee- works on. For each member, determine the owner in the set type project-works on. Go the other way around in case we need to find all employees working on a project. This is an example of procedural DML.

Furthermore, CODASYL provides no support for recursive set types or set types that only have one record type. A dummy record type needs to be introduced which is then defined as a member in two set types each having as owner record type the record type of the recursive relationship. This again implies extensive navigation when manipulating the data.
As for the previous model, no relationship types with a degree >2 is supported.

Finally, the CODASYL model allows to **logically ordering** the member records. An example could be alphabetically or based on birth date. This can be useful for data manipulation.

4. Entity-Relationship (ER) model

This model was introduced and formalized by Chen in 1976. It is one of the most popular data models for data conceptual modeling. It has **three building blocks**.

- **Entity types :** They are similar to record types in the hierarchical or CODASYL model. The basic object that the ER model represents is an entity, which is a thing in the real world with an independent existence. An entity may be an object with a physical existence (Ex : a particular person, car, house or employee) or it may be an object with a conceptual existence (Ex : a company, a job, or a university course. In other words, it represents a business concept with an unambiguous meaning to a particular set of users.
  Ex: A student, a product, a supplier, an employee …

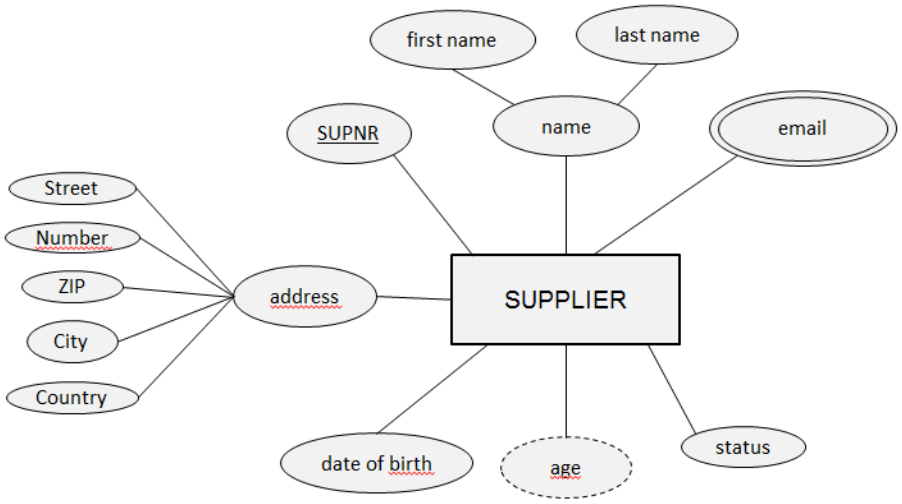
- An **entity type** defines a collection of entities that have the same attributes and represent similar business concepts. Entities are said to be instances of that particular entity type.

- **Attribute types :** Each entity has attributes, the particular properties that describe it.
  Ex : an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job. A particular entity will have a value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

- An **attribute type** defines a collection of similar attribute values. Attribute values are said to be instances of that particular attribute type.

  Several types of attributes occur in the ER model :
  - **Composite attributes** can be divided into smaller subparts, which represent more basic attributes with independent meanings. For instance, the address attribute of the EMPLOYEE entity can be subdivided into: street address, city, state, and Zip. Composite attributes can form a **hierarchy**; Ex: Street_address can be further subdivided into three simple component attributes: Number, Street, and apartment number. The value of a composite attribute is the concatenation of the values of its component simple attributes.
  - Attributes that are not divisible are called **simple** or **atomic** attributes.
  - Most attributes have a **single value** for a particular entity; such attributes are called **single-valued**. Ex: age is a single-valued attribute of a person.

- In some cases an attribute can have a set of values for the same entity. Ex: Colors attribute for a car. Cars with one color have a single value; whereas two-tone cars have two color values. Similarly, one person may not have a college degree, another person may have one, and a third person may have two or more degrees. Such attributes are called **multivalued**. A multivalued attribute may have lower and upper bounds to constrain the number of values allowed for each individual entity.
- **Derived attributes**: in some cases, two or more attribute value are related, for example, the Age and Birth_date attributes of a person. For a particular person entity, the value of Age can be determined from the current date and the value of that person's Birth date. Hence, the age attribute is called a derived attribute and is said to be derivable from the Birth date attribute which is called a stored attribute.
- **Stored attributes**: in previous example, the birth date attribute is called a stored attribute.
- A **key attribute** types is an attribute type whose values are distinct for each individual entity. In other words, in can be used to uniquely identify each entity. Ex: SUPNR, which is unique for each supplier, PRODNUR, which is unique for each product, the SSN, which is unique for each employee. The key attribute type can also be a combination of attribute types. Ex: a flight identified by a flight number. But this number is represented each day for a particular flight. Therefore, the departure date is need to identify the entities.
  = uniqueness constraint



We have an **entity type** supplier. Entity types are represented as **rectangles** in ER models. **Attribute types** are represented using **ellipses**. Address is modeled as a composite attribute type consisting of Street, number, zip, city and country. SUPNR is underlined since it is the key attribute type. The dubble lined indicate that email is multivalued. The dash lined indicate that age it is a derived attirbute type calculated base on birth-date.

- **Relationship types :** A relationship represents an association between two or more entities**.** Ex: a supplier supplying a set of products. A relationship type defines a set of relationships among instances of 1, 2 or more entity types.

Relationship types are represented using a rumble symbol in the ER model. Basically the rhombus can be represented with two adjacent arrows pointing to each of the entity types.

- The **degree** corresponds to the number of entity types participating in the relationship type.
  - A relationship type of degree one is called **unary**.
  - A relationship type of degree two is called **binary**.
  - A relationship type of degree three is called **ternary**.

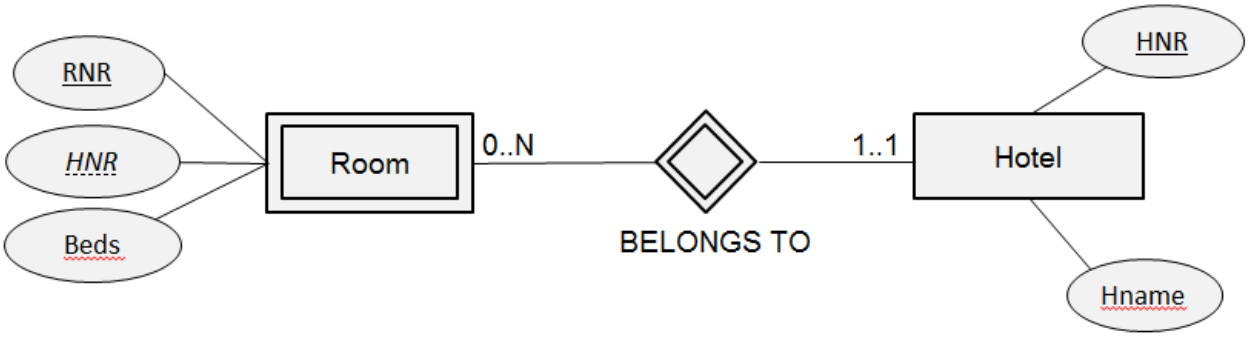
- The roles of a relationship type indicate the various directions of interpreting a relationship type. Ex: adding in an arrow "Supervises" and "supervised by".

- **Cardinality ratios** specify the minimum/maximum number of relationship instances that an entity can participate in.
  The minimum cardinality can either be zero or one. If it is zero it implies that an entity can occur without being connected to another entity. This can be referred to as partial participation. In case the minimum cardinality is one, the entity must always be connected to at least one other entity. This is referred to as total participation or existence dependency since the existence of this entity depends on the existence of another.
  The **maximum cardinality** can either be one or n. In case it is one, an entity can be connected to at most one other entity. In case the maximum is N, an entity can be connected to at most N entities. Relationships types are often characterized by means of their maximum cardinalities.
  For binary relationships, there are 4 options :
  - 1:1
  - 1:N
  - N:1
  - M:N

Relationship types can also have attribute types, similar to those of entity types. These can be migrated to one of the participating entity types. Attribute types belonging to M:N relationship types cannot be determined by a single entity type. They can only be specified as relationship attribute type.

Weak entity types are a special class of entity types. These are entity types that do not have key attribute types of their own. Entities belonging to a weak entity type are identified being related to specific entities from another entity type (the owner entity type) in combination with some of their own attributes (called partial keys). A weak entity type is always existence depend from this owner entity type. An existence depend entity type does not imply a weak entity type.
Ex: A hotel administration. Hotel has an HNR and Hotel name. Every hotel has a unique hotel number. Hence HNR is the key attribute. A room is identified by a room number and a number of beds. Within a particular hotel, each room has a unique room number but a room number can appear for multiple rooms being in different hotels. Hence RNR does not suffice the key attribute. The entity type room is therefore a weak entity type since it cannot produce its own attribute type. It

needs to borrow HNR from hotel to come up with a key attribute type which is now a combination of RNR and HNR.



In contrast, regular entity types that do have a key attribut are called **strong entity types**.

- The ER model has an attractive and user-friendly graphical notation. Hence, it has the ideal properties to build a conceptual scheme or model.

A **domain** specifies the set of values that may be assigned to an attribute for each individual entity. A domain gender can be specified as having only two values: male and female. A date domain allows finding dates as days, followed by months, followed by years. By convention, domains are not displayed in ER diagrams. A domain can also contain **Null values**, which means the value is not applicable, not relevant or not known. It is thus not the same as the value zero. Ex: domain email address which allows for null values in case the email address is not known.

5. Enhanced entity-Relationship (EER) model

The EER model is an extension of the ER model. It includes all the modeling concepts of the ER model. In addition, it incorporates the following additional semantic data modeling concepts.

- **Specialization** is the process of defining a set of subclasses of an entity type. This entity type is called the **superclass** of the specialization. The set of subclasses that form a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass.
  Ex: The set of subclasses {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE. We may have several specializations of the same entity type based on different distinguishing characteristics. In other words there exist other specializations of the employee base on method of pay for example.

  The specialization process defines a subset of subclasses of an entity type representing an "IS A" relationship. It can then establish addition specific attribute types for each subclass. Ex: a student can have a master thesis; a professor can have an academic degree attribute type. During the specialization, it is also possible to establish additional specific relationship types between each subclass and other entity types. Ex: a student can register for courses, a professor can teach courses.
- **Generalization**, also called abstraction, is the revers process of specialization. **Specialization** corresponds to a top-down process of conceptual refinement whereas **generalization** corresponds to a bottom-up process of conceptual synthesis. In this process, we suppress the differences among several entity types, identify their common features, and generalize them

into a single superclass of which the original entity types are special subclasses. Ex: if we consider the entity types Car and Truck. Because they have several common attributes, they can be generalized into the entity type vehicle.
We use the term generalization to refer to the process of defining a generalized entity type from the given entity type.

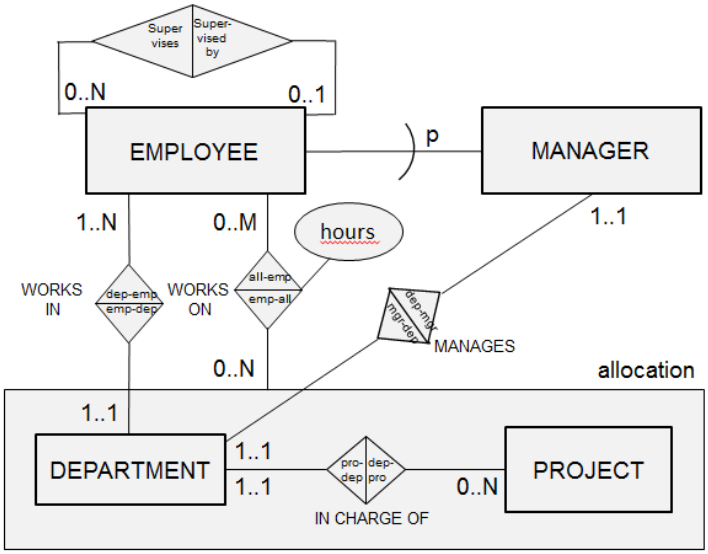
- **Disjointness constraint**
  - **Disjoint**: an entity can be member of at most one of the subclasses.
  - **Overlap**: the same entity may be member of more than one subclass.
- **Completeness constraint**
  - **Total**: specialization where every entity in the superclass must be a member of some subclass.
  - **Partial**: allows an entity not to belong to any of the subclasses and to only belong to the super classes.

- The disjointness and completeness constraints are **independent** which leads to many possible combinations.
  - Disjoint and total
  - Disjoint and partial
  - Overlapping and total
  - Overlapping and partial

A specialization can be several levels deep. In other words, a subclass can be a superclass of another specialization. In a **specialization hierarchy**, every subclass can only have a single superclass. In a **specialization lattice**, a subclass can have multiple super classes. In both cases, a subclass inherits the attribute types and relationship types of all its predecessor super classes all the way to the root of the hierarchy.
The concept where a shared subclass (i.e a subclass with multiple parents) inherits from all of its parents is called multiple inheritances.

- **Categorization**: A category is a subclass that has several possible super classes. Each superclass represents a different entity type. The category represents a collection of entities that is a subset of the union of super classes. Ex: Account holder is the subset of a union between person and company.
  Inheritance in this case corresponds to an entity inheriting only the attributes and relationships of that superclass it is a member of **(= selective inheritance).**
  It can be total or partial. In a total categorization all entity of the super classes belongs to the subclass. A total categorization can also be represented as a specialization or generalization.

- **Aggregation:** The idea is that entity types that are related by a particular relationship type can be combined into a higher-level aggregate entity type. This can be useful when the aggregate entity type has its own attribute types and relationship types.
  Ex**:** Consultant working on projects can be combined into the aggregation of “participation”.



We partially specialized employee into manager. Then we connect the manager subclass to the department entity type. Department and project have been allocated into allocation. This then participated in the relationship type “works on” with EMPLOYEE.

6. Procedure to design EER models

An EER model can be designed by following these **steps**:

- Identify the entity types
- Identify the relationship types and assert their degree
- Assert cardinality ratios and participation constraints
- Identify the attributes and assert whether they are simple/composite; single/multiple valued;
- Link each attribute type with an entity type or a relationship type
- Denote the key attribute type(s) of each entity type
- Identify weak entity types and their partial keys
- Apply abstractions such as generalization/specialization, categorization and aggregation
- Assert the characteristics of each abstraction: disjoint/overlapping, total/partial
- Document semantics that cannot be represented in the (E)ER schema as separate "business rules"

Although the EER model offers some new interesting modeling concepts (specialization, generalization, aggregation and categorization), the **limitation** of the EER model still apply …

- **Temporal aspects** can still not be modeled.
- **Consistency** among multiple relationship types cannot be enforced.
- **Integrity** rules or behaviors cannot be specified.

7. Relational model

7.1 Introduction

The relational model is one of the most popular data models in used nowadays. It was first formalized in 1970. This model is a formal model with a mathematical foundation based on set theory and first order predicate logic. It has no graphical representation which makes it less suitable to be used as a conceptual model.

It consists of the following **basic concepts**.

- The relational model represents the database as a **collection of relations**
- A relation is defined as a **collection of tuples** that each represents a similar, real world entity such as product, supplier, employee, etc.
- A **tuple** is an ordered list of (attribute) values that each describes an aspect of this entity such as supplier number, name, address, etc. Each tuple is uniquely identified by its **primary key**. Besides, tuples are interrelated by means of **foreign keys**.
  - ⇨ Various commercial implementations exist, provided by Oracle, IBM, Microsoft, etc.

7.2 The relational model

A relation can also be interpreted as a table of values. Let's take for example a relation called SUPPLIER. A relation corresponds to an entity type from the EER model. Each tuple corresponds to a row in a table or to an entity in the EER model. Attribute types can be seen as column names. These are identical to the attribute types in the EER model (Ex: supplier number, name, address, city, status, etc.). Each attribute type value corresponds to a single cell.

A **relation schema** R, denoted by $R(A_1, A_2, A_3, \ldots, A_n)$ is made up of a **relation name R** and a list of **attribute types (A_1, A_2, A_3, …, A_n).** Each attribute type A_i is the name of a role played by some domain D_i in the relation schema R.
D_i is called the domain of A_i and is denoted by $dom(A_i)$. A gender domain can specify the values male and female. A time domain can specify time as day followed by month followed by year. R is the name of the relation schema. It is recommended to use a meaningful name.
Ex:

Student (SN, Name, HomePhone, Address)
Professor (SSN, Name, HomePhone, OfficePhone, e-mail)
Course (CourseNo, CourseName

A domain can be used multiple times in a relation scheme. An advantage of using a domain is that if definition would ever have to changed, then this changed should only be done in the domain definition. Therefore, this improves the maintainability of the model.

A relation also, called extension, r(R) for the relation schema $R(A_1, A_2, A_3, \ldots, A_n)$ is a set of m-tuples $r=\{ t_1, t_2, t_3, \ldots, t_m\}$. each tuple is an ordered list of n values $t = <v_1, v_2, v_3, \ldots v_n>$, corresponding to a particular entity such as a particular student, supplier, etc. Each value v_i, $1 \leq i \leq n$, is an element of $dom(A_i)$ or is a special *null* value. NULL value means that the value is **missing**, **irrelevant** or **not applicable**.

Ex: Student(100, Michael Johnson, 123 456 789, 532 Seventh Avenue)
Professor(50, Bart Baesens, 987 654 321, 876 543 210, Bart.Baesens@kuleuven.be)
Course(10, Principles of Database Management)

More formally, A relation r(R) is a mathematical relation of degree n on the domains dom(A_1), dom(A_2), dom(A_3), …, dom(A_n) and is a subset of the Cartesian product of the domains that define R:

$$r(R) \subseteq (dom(A_1) \times dom(A_2) \times dom(A_3) \times \ldots \times dom(A_n))$$

The **Cartesian product** specifies all possible combinations of values from the underlying domains. Of all these possible combinations, the current relation state represents only the valid tuples that represent a particular state of the real world.

It is important to know that a representation essentially represents a set. Hence, there is no logical ordering of tuple sin a relation. The relation also does not have any duplicate tuples. There is however an ordering on tuples based on how the relation was defined. According to the relational model, each attribute type is single valued and atomic. Therefore, no composite or multivalued attribute types are allowed.

7.3 The constraints

The relational model allows defining various constraints on the values of the attribute types.

7.3.1 Domain constraints

Domain constraints state that the value of each attribute type A must be an atomic and single value from the domain dom(A).

Ex: COURSE(coursenr, coursename, study points)

Right: (10, Principles of Database Management, 6)
Wrong: (10, (Principles of Database Management, Database Modeling), 6). It specifies two values for the course name.

7.3.2 Key constraints

To specify the key and uniqueness constraints, it seems important to first explain the concept of a key. A relation is a set of tuples. All tuples must therefore be distinct. No two tuples can have the same combination of values for all their attribute types. A super key is defined as a subset of attribute types of a relation schema R with the property that no two tuples in any relation state should have the same combination of values for these attribute types. A super key specifies a uniqueness constraint in the sense that no distinct tuples in a stat can have the same value for the super key.

Every relation has at least one default superkey, the set of all its attribute types. A super key can have redundant attribute types. Ex: {StudentNr, StudentName, StudentBirthdate} is a superkey, but note that both StudentName and StudentBirthdate are redundant.
A **key K** of a relation scheme R is a super key of R with the additional property that removing any attribute type from K leaves a set of attribute types that is no super key of R. A key does not have redundant attribute types; hence it is also called a minimal super key. For a student relation, Student number is a key.

In general, a relation scheme may have more than one key. For instance, for a product relation, a product can have a unique product number and unique product name. Each of these keys is called a **candidate key**. One of them is designated as the **primary key** of the relation. This primary key will be used to identify tuples in the relation and establish connections in other relations. It can also be used for storage purposes and define indexes in the physical scheme. The other candidate keys are then referred to as candidate keys.

7.3.3 Entity integrity

The attribute types that make up the primary key must always satisfy the NOT NULL constraint. Otherwise it would not be possible to identify some tuples. This is the entity integrity constraint. A NOT NULL constraint can also be defined for other attribute types such as the alternative keys.

A relational database schema S is a set of relation schemas $\{R_1, R_2, R_3, \ldots, R_m\}$ and a set of *integrity constraints* IC.
A relational database state DB of S is a set of relation states $\{r_1, r_2, r_3, \ldots, r_m\}$ such that each r_i is a state of R_i and such that the relation states satisfy the integrity constraints specified in IC.
The DBMS will have to take care that the integrity constraints are always checked and the violations reported if the database state is updated.

7.3.4 Referential integrity constraints

Similar to relationship types in the EER model, also relations in the relational model can be connected. These connections are established thanks to a foreign key. A set of attribute types FK in a relation schema R_1 is a ***foreign key*** of R_1 if two conditions are satisfied.

- The attribute types in FK have the same domains as the primary key attribute types PK of a relation R_2;
- A value FK in a tuple t_1 of the current state r_1 either occurs as a value of PK for some tuple t_2 in the current state r_2 or is NULL.

⇨ The conditions for a foreign key specify a **referential integrity constraint** between two relation schemas R_1 and R_2.

7.3.5 Semantic integrity constraints

The relational model has its shortcomings. These are more general constraints, which cannot be enforced in the standard relation mode. Extensions have been developed to the relational model to enforce some of these constraints e.g. stored procedures or triggers.

7.4 Normalization

7.4.1 Informal normalization guidelines

Minutes 22-24 explanation drawback of unnormalization

In order to have a good relational database schema, all relations in the schema should be normalize-ed. Relational schemas that were generated from an (E)ER schema will automatically have this property, if the translation rules were applied correctly. On the contrary, if the relational schema was not derived from an (E)ER schema, a formal normalization mechanism can be applied to transform the relational schema to a normalized form.

Advantages :

- At the **logical level**, the users can clearly understand the meaning of the data and formulate correct queries.
- At **the implementation level**, storage space is used efficiently and the risk of inconsistent updates is reduced.

Important to design the relational scheme in such a way that it is easy to explain its meaning. Next, attribute types from multiple entity types should not be combined in a single relation. It should also be make sure to mitigate the risk of insertion, delete and update anomalies when designing the relational scheme. Finally, excessive amounts of NULL values in a relation should be avoided.

- **Functional dependency X -> Y**, between two sets of attribute types X and Y implies that a value of X uniquely determines a value of Y. if X -> Y in R, this does not say whether or not Y->X in R.

Examples

- ⇨ SSN → Employee NAME (Note: not necessarily other way around!). Employee name is functionally dependent from SSN. In other words, the SSN uniquely determines the employee name. The other way round isn't necessarily true since multiple employees can share the same name.
- ⇨ PNUMBER → {PNAME, PLOCATION}. Project number uniquely determines a project name and project location. Therefore, PNAME and PLOCATION are functional dependent from PNUMBER.
- ⇨ {SSN, PNUMBER} → HOURS

If X is a candidate key or F, this implies that X -> Y for any subset of attribute types Y or R.

- **Prime attribute type** is an attribute type that is part of a candidate key.

  ⇨ **Normalization** of a relational scheme is a process of analyzing the given relation schemas based on their functional dependencies and candidate keys to **minimize redundancy** and **insert**, **deletion** and **anomalies**.
  Unsatisfactory relation schemas that do not meet the normal form tests are decomposed into smaller relation schemas.

7.4.2 First normal form (1NF)

The first normal form states that every attribute type of a relation must be **atomic** and **single valued**. This means there cannot be any composite or multivalued attribute types. In case of a multivalued attribute type, remove it and put it in a separate relation, along with the primary key of the original relation as a foreign key. The **primary key** of the new relation is than the **combination** of the **attribute type and the primary key** of the original relation. Composite attribute types need to be decomposed in their parts.

Ex: The relation is not in first normal form since DLOCATION is a multivalued attribute type. Therefore, we should remove DLOCATION from the relation DEPARTMENT and put it together with DNUMBER as a foreign key. The primary key is then the combination of both: DNUMBER and DLOCATION.

DEPARTMENT(DNUMBER, DLOCATION, *DMGRSSN*)

⇩ 1 NF

DEPARTMENT(DNUMBER, *DMGRSSN*)
DEP-LOCATION(*DNUMBER*, DLOCATION)

Assumptions: A department can have multiple locations! Multiple departments are possible at a given location!

7.4.3 Second normal form (2NF)

- A functional dependency X → Y is a **full functional dependency** if removal of any attribute type A from X means that the dependency does not hold anymore.
  Example: SSN, PNUMBER → HOURS; indeed, to know the number of hours an employee worked on a project, we need both the SSN and the project number.
  PNUMBER → PNAME
- A functional dependency X → Y is a **partial dependency** if some attribute type A ∈ X can be removed from X and the dependency still holds.
  Example: SSN, PNUMBER → PNAME. It only depends on PNUMBER and not on SSN

  A relation R is in **2NF** if it satisfies 1NF and every nonprime attribute type A in R is fully functional dependent on any key of R. As a reminder, a prime attribute type is an attribute type that is part of a candidate key.
  If this is not the case, **decompose** it and set up a new relation for each partial key with its dependent attribute types.
  Furthermore, it is also important to make sure to keep a relation with the original primary key and any attribute types that are fully functional dependent on it.

Ex: R1 is in 1NF since there are no multivalued and no composite attribute types. However, it is not in 2NF. The attribute type PNAME is not fully functional dependent on the primary key. Indeed, it only depends on PNUMBER. On the contrary, the attribute type HOURS is fully dependent from SSN and PNUMBER. Therefore, we need to create a new relation.

R1(SSN, PNUMBER, PNAME, HOURS)

R11(*SSN, PNUMBER*, HOURS)
R12(PNUMBER, PNAME)

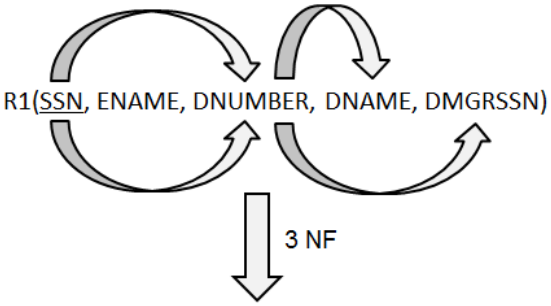
Assumptions:
- An employee can work on multiple projects
- Multiple employees can work on a project
- A project has a unique name

7.4.4 Third normal form (3NF)

A functional dependency X → Y in a relation R is a **transitive dependency** if there is a set of attribute types Z that is neither a candidate key nor a subset of any key of R, and both X → Z and Z → Y hold.

A relation is in **3NF** if it satisfies 2NF and **no** nonprime attribute type of R is transitively dependent on the primary key.
If this is not the case, decompose and set up a relation that includes the non-key attribute types that functionally determine other non-key attribute types.

The relation R1 contains information about employees and departments. The SSN attribute type is the primary key of the relation. We have 2 transitive dependencies in R1. DNAME is transitively dependent from SSN via DNUMBER. In other words, DNUMBER is functionally dependent from SSN and DMGRSSN is functionally dependent from DNUMBER. Likewise, the DMGRSSN is transitively dependent from SSN via DNUMBER. To bring this in 3NF we remove the attribute type DNAME and DMGRSSN and put it in a new relation R12 with DNUMBER as primary key.
R11 can be called EMPLOYEE and R12 DEPARTMENT.



R11(SSN, ENAME, *DNUMBER*)
R12(DNUMBER, DNAME, *DMGRSSN*)

Assumptions:
- An employee works in 1 department
- A department can have multiple employees
- A department has 1 manager

7.4.5 Boyce-Codd normal form

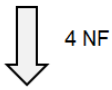
A functional dependency X → Y is called **trivial** if Y is a subset of X
Ex: SSN, NAME → SSN.
A relation R is in **Boyce-Codd** normal form if and only if, for every one of its non-trivial functional dependencies X → Y, X is a superkey; that is, X is either a candidate key or a superset thereof.

Boyce-Codd normal form is said to be stricter than 3 NF. Hence, every relation in Boyce-Codd normal form is in 3NF. Hence, a relation in 3 NF is not necessarily in BCNF

7.4.6 The fourth normal form

There is a **multi-valued dependency** from X to Y
X →→ Y if and only if each X value exactly determines a set of Y values, independently of the other attribute types
A relation is in 4NF if it is in Boyce-Codd normal form and for every one of its non-trivial multivalued dependencies X →→ Y, X is a superkey—that is, X is either a candidate key or a superset thereof
This normal form is often neglected in database modeling.

R1(course, instructor, textbook)



R11(course, textbook)
R12(course, instructor)

Multivalued dependency between course and textbook. In other words, each course exactly determines a set of textbooks, independently from the instructor. We create two relations.

Assumptions:
- A course can be taught by different instructors
- A course uses the same set of textbooks for each instructor

8 Unified Modeling language

There is a need of some standard approach to cover the entire spectrum or requirements analysis, modeling, design, implementation, and deployment of databases and their applications. One approach that is receiving wide attention and that is also proposed as a standard by the object Management Groups is the **Unified Modeling Language (UML)**.

Object-oriented data models are typically represented using **UML**. As the term suggest, this is essentially an OO system modeling notation. It does not only focus on data requirements but also on process modeling. It was accepted in 1997 as a standard by the OMG and approved as ISO standard in 2005. The most recent version was introduced in 2015 and is known as UML 2.5. To model both the data and process, UML foresees multiple diagrams such as use case diagrams, sequence diagrams, package diagrams, deployment diagrams, etc. from a database modeling perspective, the class diagram is the most important as it allows to visualize both class and their associations.

Recap of object-orientation:

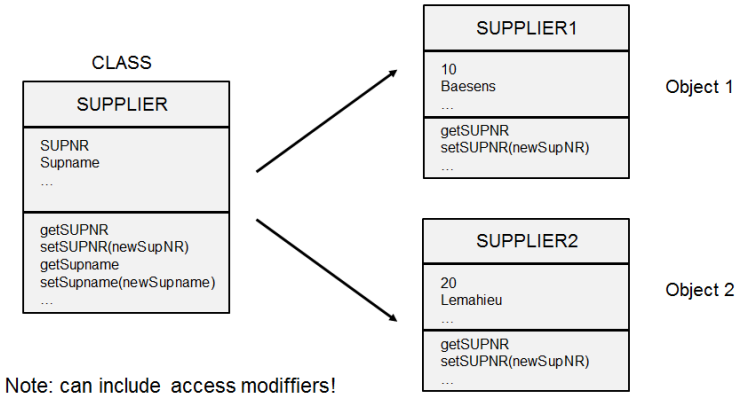
The **two important building blocks** are:

- **Classes**: blue print building definition for a set of objects. A class in OO corresponds to an entity type in ER. Ex: Student
- **Object**: instance of a class. An object corresponds to an entity. Ex: Student Ysaline. Each object is characterized by both **variables** (name, gender, birth date) and **methods** (calcAge(), isBirthday(), etc.).

Information hiding states that the variables of an object can only be accessed through either getter or setter methods. **Getter** methods allow retrieving the value of a variable whereas the **setter** method allows setting it. The idea is to provide protective shields to make sure that values are always correctly retrieved or modified.

Inheritance is here also supported. Therefore, a superclass can have one or more subclasses which inherit both the variables and methods from the superclass. Ex: student and professor can be a subclass of the person superclass.

In OO, **method overloading** is supported which means that methods in the same class can have the same name, but a different number or type of input arguments.



Ex: Class: SUPPLIER. A class is represented as a **rectangle** with **three sections**. In the **upper part**, the name of the class is mentioned, in the **middle part** the variables and in the **bottom part**, the methods. UML also allows defining **access modifiers** for each of the variables and methods. This access modifier specifies who can have access to the variable or method.

- **Private**: variable and methods can only be accessed by the **class itself**.
- **Public**: in case variables and methods can be **accessed by any other class.**
- **Protect**: variable and methods can be accessed by **both the class and sub-class**.

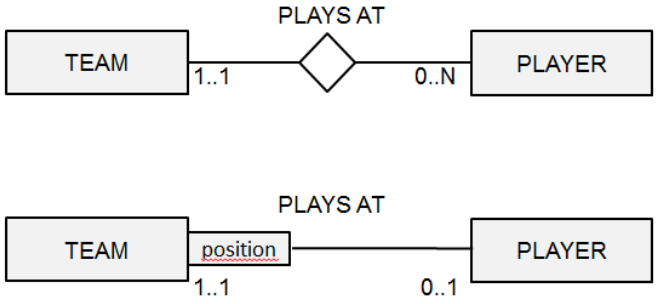
⇨ To enforce the concept of information hiding, it is recommend declaring all the variables as private and accessing those using getter and setter methods.

Classes can be related using **association types**. Multiple associations can be defined between the same classes. Besides, also unary or reflexive associations are possible where class engages in a relationship with itself. Associations can be further augmented with directions reading arrows which specify the direction of querying and navigating through it. In a unidirectional association, there is only one way of navigating as indicated by the arrow whereas in the binary association, both directions are possible and hence, there is no arrow.

UML model		EER model	
Class		Entity type	
Object		Entity	
Variable		Attribute type	
Method		-	
Association		Relationship type	
Link		Relationship	
Multiplicity	*	Cardinality	0..n
	0..1		0..1
	1..*		1..n
	1		1..1

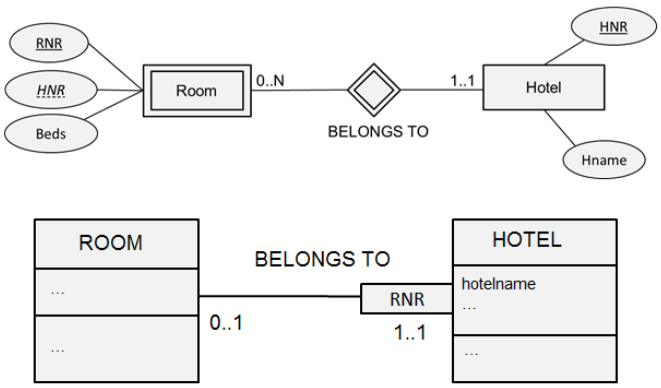
⇨ The asterisk is introduced to denote a maximum cardinality of n.

A qualified association is a special type of association. More specifically, he uses a qualifier to further refine the association. The qualifier specifies 1 or more variable those are used as index key for navigating from the qualified class to the target class. It allows reducing the multiplicity of the association because of this extra key.

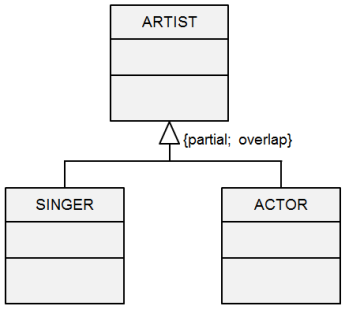


2 classes : team and player. They are connected using a 1:n relationship type since a team can have 0:N player and a player is always related to exactly one team. This can be represented using a qualified association by including the position variable as the index key or qualifier. A team at a given position has 0 to N player and a player always belongs to one team.

⇨ Qualified associations can be used to represent weak entity types.



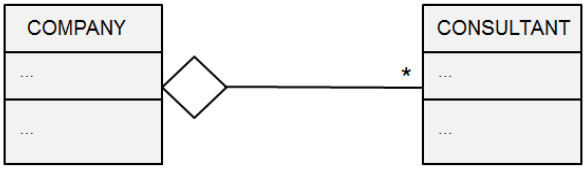
Similar to the EER model, the UML model also allows to model specialization or generalization relationships.



We start from our earlier example with artists who can be singers and actors. The **hollow triangle** represents a specialization in UML. The specialization characteristics such as total, partial, disjoint and overlap can be added next to the triangle. Besides, it is important to mention that UML also supports multiple inheritances.

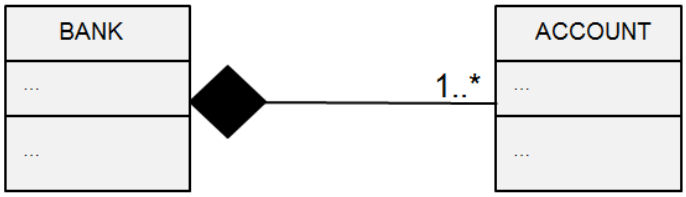
Two types of aggregations are possible in UML :

Shared aggregation: the part object can simultaneously belong to the multiple composite objects. In other words, the **maximum multiplicity** at composite side is **undetermined**. The part object can also occur without belonging to a composite object. The shared aggregation does represent a rather loose coupling between both classes.



The shared aggregation is indicated by a **hollow diamond**. Shared aggregation between company and consultant. A consultant can work for many companies. When a company is removed, any consultant that worked for it remains in the database.

Composite aggregation: the part object can only belong to one composite. The maximum multiplicity at composite side is 1 and the part object will be automatically removed when the composite object is removed.



The composite aggregation is indicated by a filled diamond. Composite aggregation between bank and account.

When a bank is removed, all connected account objects disappear as well.

UML offers various advanced modeling concepts which allow furthering a semantic store data model. The **changeability property** specifies the type of operations which are allowed on either variable values or links.

- **Default**: allows any type of edit.
- {**addOnly**}: only allows additional values or links to be added. So no deletions. Ex: for a purchase order, can only be added and not removed for a given supplier.
- {**Frozen**}: allows no further changes once the value or link is established. Once a value has been assigned, it can no longer change.

The **object constraints language** or OCL which is also part of the UML standard, allows to specify various types of constraints. The OCL constraints are defined in a declarative way. They specified what should be true but not how this should be accomplished. In other words, no control flow is

provided. They can be used for various purposes: specify invariants for classes specify pre- and post-conditions for methods, used as a navigation language, define constraints on operations.

A **class invariant** is a constraint which holds for all objects of a class. Ex: constraint specifying that a Supplier status should be bigger than 100.

Pre and post-conditions on methods must be true when a method either begins or ends. Ex: before the method withdraw has been executed, the balance must be positive. After it has been executed, the balance must still be positive.

- ⇨ OCL also allows defining more **complex constraints**. It is a <u>powerful</u> language which allows adding a lot of semantics to the model.

Dependency is using relationship that states that a change in the specification of a thing may affect another thing that uses it. It is denoted by a dashed line in the UML diagram. Ex: when an object of one class uses an object of another class in its methods but the referred object is not stored in any variables.

Here, we have 2 classes: EMPLOYEE and COURSE. An employee can follow courses as part of a company education program. The employee class includes a method to course to see if an employee follows a particular course. Hence, an employee object makes use of course object in one of its methods. This explains the dependency between both classes.

CHAPTER 4 : DATABASE DESIGN

1. Database design process

Let's first start by zooming out and see where we are in the database design process … We started by **collecting** and **analyzing** the data requirements which we referred to as the universe of discourse. These requirements where then **formalized into a data model**. As our conceptual data model, we can make use of either EER model or OO model since both have a user friendly graphical representation. As the next step, we can map our conceptual data model to a logical date model. This data model is based upon the data model of the DBMS that will be used for the actual implementation. This chapter does explain how to map EER or OO models to CODASYL, relation or OO models. Finally, a physical data model is the actual implementation of the logical data model for example in Microsoft or in Oracle.

2. Conceptual database design

The conceptual data model needs to satisfy various **requirements**.

- It needs to be able to accurately and unambiguously model the universe of discourse.
- It must be **flexible** such that new or changing requirements can be easily added to the model.
- The model must be **user friendly** and easy to understand the graphic representation so that it can be used as a communication tool between the database developers and the business users.
- **DBMS or implementation independent** since its only goal is to collect and analyze data requirements.

- When considering these requirements, it becomes logical that only two models are illegible for a conceptual data model: EER model and OO model. Since both of these models also have their shortcomings, it is of key importance to document.
  As a reminder, EER models are temporary snapshot. Hence, temporal aspects cannot be included and need to be written down as business rules. Ex: an employee cannot return to a department.
- To sum up, the conceptual data model needs to be complemented with some **business rules** which can then be reconsidered during the actual implementation.

Key steps when developing a conceptual data model:

- Identify and name entity types
- Identify and name relationship types
- Assign role names and cardinalities to relationship types
- Identify and assign attribute types and assigned to either the entity or relationship type.
- Define key attribute types
- Identify weak entity types

- Apply EER constructs such as specialization, categorization and aggregation

It is important that these activities are executed interactively in close collaboration with the business user. When naming the modeling concepts, it's important the use conventional naming concepts to facilitate the model understanding and the future maintenance.

The steps to design an OO model are quite similar to the ones listed before for the conceptual model.

- Identify and name (persistent) classes
- Identify and name association types
- Assign role names, cardinalities and direction reading arrows to association types
- Identify and assign attribute types
- Define key attribute types
- Identify, name, design and assign methods
- Identify weak entity types
- Identify specialization (is-a) and aggregation (part of) relationships

- Also here, additional semantics can be written down as business rules.

3. Logical database design

A logical data model is defined in terms of the data model adopted by the DBMS package which will be used for the actual implementation. Regarding the software environment available, we will map a conceptual model to a CODASYL, relational or OO model. This mapping can be done automatically by a database modeling tool. Depending on the conceptual and logical data model selected, it is possible that we lose or add some semantics. They should be carefully documented, once again.

3.1 Mapping an EER model to a CODASYL model

This is an example where our conceptual model is richer, in terms of semantics, than our logical model. Hence, we will use semantics during the mapping.
As a reminder, let's briefly explain **the two key concepts** of the CODASYL model.

- A **record type** represents a set of records that consists out of data items. Vectors can be defined for multivalued attribute type and repeated groups can be used for composite multivalued attribute type. It can be owner and member in multiple set types.
- A **set type** describes a 1:N relationship between an owner record type and a member record type.

To map an EER model to a CODASYL model, we start by **mapping every entity type** to a **record type**. Atomic or composite attribute types can be directly supported in the CODASYL model. Multivalued attribute types can be modeled by vectors or repeated groups. For each weak entity type, we create a separate record type defined as member in a set type with as owner the record type on which it is existent dependent. Every 1:N relationship type can be translated to a set type or by the owner or member record type are determined according to the EER cardinalities.

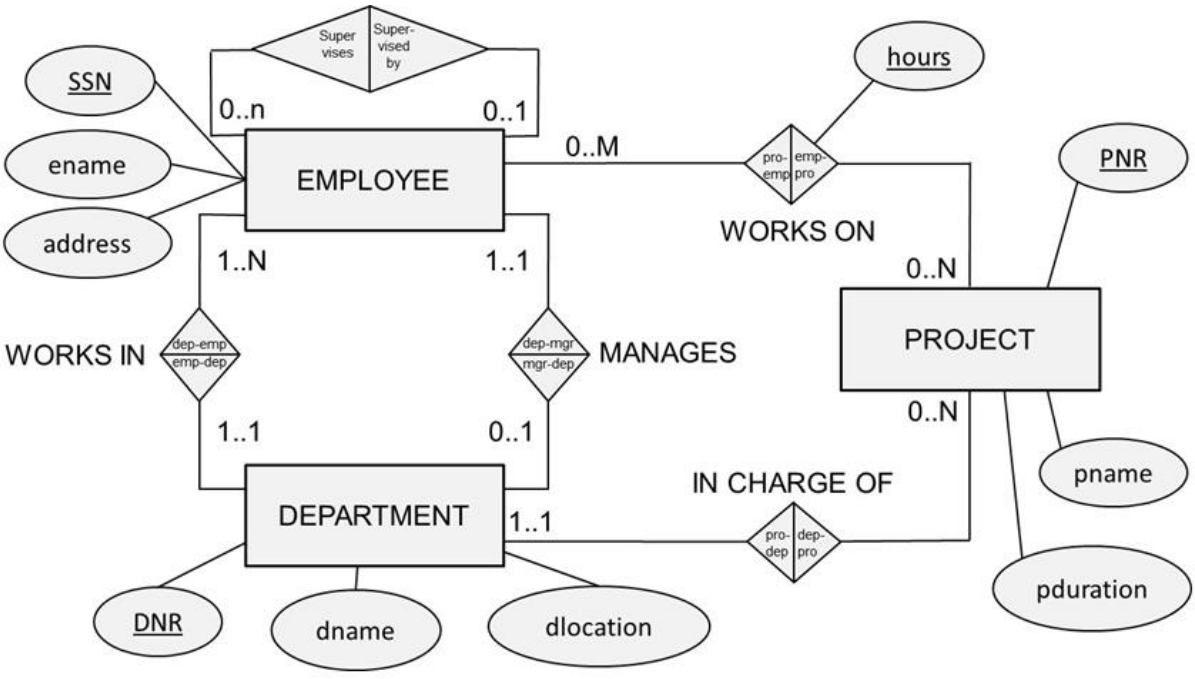
A binary 1:1 relationship type is also modeled using a set type. The owner and member are determined arbitrarily or via existence dependency. We are losing semantics here as we cannot enforce one record member in every set.

- Binary N:M relationship types are not supported in the CODASYL model. Hence, we have to implement them by including a dummy record type.
  Dummy record types are defined as a member in the 2 sets type and they contain attributes of the relationship type.
- Recursive relationship types are also mapped using an extra dummy record type. It then becomes a member in two set types. Both model an explosion and implosion relationship.
- Ternary relationship types are also not supported in the CODASYL model. Therefore, they need to be implemented by introducing dummy record types which becomes member in three set types.

EER constructs such as specialization, categorization and aggregation are not possible in the CODASYL model and need to be reconstructed using set types which obviously leads to losing semantics.
Ex: superclass or subclass construct. We could create a record type for superclass that becomes owner in set types with subclasses as members. This cannot guarantee that each set contains at most 1 member and cannot indicate the type of specialization (partial or complete, overlap or disjoint). Besides, we can also opt to use one record type that contains the data from the superclass and all the subclasses. A drawback is that it results into loads of empty data fields.

Ex: HR database. The underlined cardinalities are the ones that correspond to the ER model. They cannot be enforced in the CODASYL model. We cannot enforce that a department should have at least one employee or that one department should have at least one manager.

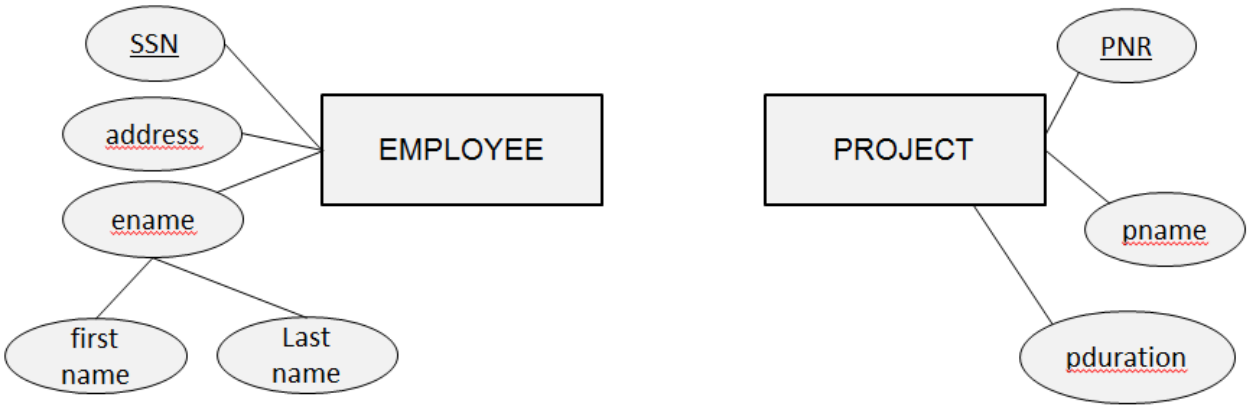
3.2 Mapping an EER model to a relational model



Let's briefly sum up the key concepts of a relational model :

- It consists out of relations.
- A relation is a set of tuples characterized by attributes.
- A relation has a primary key which uniquely identifies its tuples.
- Relationships can be established by means of foreign keys which refer to primary keys.
- All relations in a relational model are **normalised**. Such that no redundancies or umbiguities are left in the model.

The first step is to **map each entity type** into a **relation**. Simple atribute types can be directly mapped wheareas composite attribute types need to be decomposed into its component attributes. One of the key attribute types of the entity type can be set as the primary key of the relation.



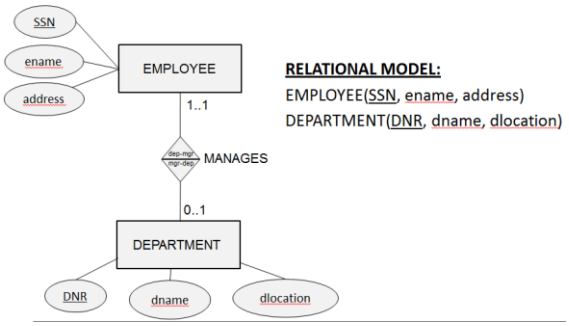
We have two entity types : employee and project. We create relations for them. The employee entity type as 3 attribute types : SSN, address (atomic attribute type), ename which is a composite attribute type consisting of first name and last name. The project entity type also has three attribute types : PNR, pname, pduration. Both key attribute types have been mapped to the primary keys of bith relations. Also note that the ename has been decomposed into first and last name in the relation EMPLOYEE.
Then we create 2 relations for each entity type particpating in a binary 1:1 relationship type. The connection can me made by including a foreign key in one of the relations to the primary key of the other. In case of existence dependancy, we put the foreign key in the existend dependent relation and declare it as NOT NULL. The attributes of the 1:1 relationship type can then be added to the relation with the foreign key.

Let's consider a "manages" relationship type between employee and department. An employee manages either 0 or 1 departement and a department is manged by 0 or 1 employee. In other words, department is existence dependent from employee. We create relations for both entity types and then add attribute types.

How to map the relationship type ?

A first option would be to add a foreign key "DNR" to the employee relation which refers to the primary key DNR in department. This foreign key can be null since not all employees manage a department.

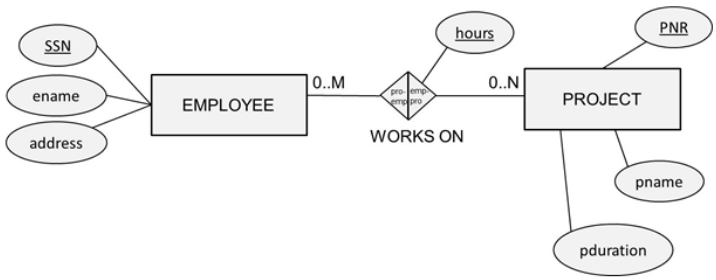
- **Binary 1:N** relationships can be mapped by using a **froreign key** in the relation corresponding to the participating entity type at the N-side of the relationship type.
- **The foreign key** refers to the primare key of the realtion corresponding to the entity type at the 1-side of the relationship tpye.
  Depending upon the minimum cardinality, the foreign key can be defined as NULL or not allowed.
- The **attribute types** of the **1:N relationship** type can be added to the relation corresponding to the participating entity type.



The “works in” relationship type is an example of a 1:N relationship type. An employee works in exactly one department whereas a department can have 1 to N employees working in it. The strating date represnet the date an employee started working in the department. We started by creating the relation EMPLOYEE and DEPARTMENT for both entity types.
There are again two options to map this relationship thype into the relational model.

- **M:N** relationship types are mapped by introducing a **new relation R**. The primary key of R is a combination of foreign keys referring to the primary keys of the relations corresponding to the participating entity types.
  The attribute types of these relationship types can also be added to R.

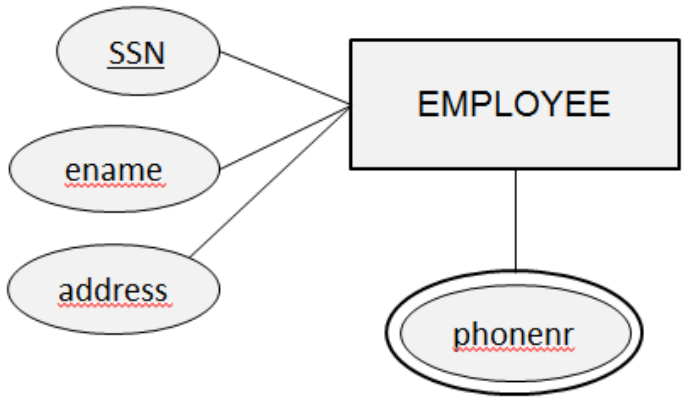
The WORKS ON relationship type is a good example. An employee works on 0:N projects whereas a project is being worked on by 0:M employees. We start by creating relations for both entity tpes. We cannot add a foreign key to the employee relation as it would give as a multivalued attribute type since an employee can work on multiple projects. Likewise, we cannot add a foreign key to the project relation as a project is been worked on by multiple employees. In other words, we need to create a new relation to map the EER relationship type “works on”.



Here we can see the relation WORKS ON defined. It has 2 foreign keys: SSN and PNR which together make up the primary key and can thus not be null. The hours attribute type is also added.

If we changed the assumptions, let’s say that an employee works on at least one project and a project is being worked on by at least one employee. In other words, the minimum cardinalities changed to one on both sides. The solution remains the same. But only 2 of the cardinalities will be enforced.

For each multivalued attribute type, we create a new relation R. We put a multivalued attribute type in R together with a foreign key referring to the primary key of the original relation. Multivalued composite attribute types are again decomposed into their components. The primary key can than be set base upon the assumptions.



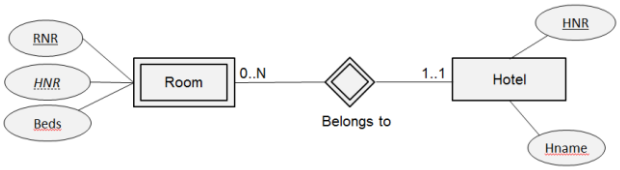
Phone number is a multivalued attribute type. Indeed, an employee can have multiple phone numbers. We create a new relation EMP-PHONE. It has two attribute types : PhoneNr and SSN. The latter is a foreign key refering to the employee relation. If we assume that each phone number is assigned to only one employee then the attribute type PhoneNr suffises as the primary key of the relation EMP-PHONE.

RELATIONAL MODEL
EMPLOYEE(SSN, ename, address)
EMP-PHONE(PhoneNr, *SSN*)

If a phone number can be shared by multiple employees, this attribute type is no longer appropriated as primary key of the relation. Also SSN cannot be assigned as primary key since an employee can have multiple phone numbers. Hence, the primary key becomes a combination of both phone number and SSN. This illustrates how the business specifics can help to find the primary key of a relation.

RELATIONAL MODEL
EMPLOYEE(SSN, ename, address)
EMP-PHONE(PhoneNr, *SSN*)

A weak entity type should be mapped in a relation type R with corresponding attributes. A foreign key must be added, referring to the primary key of the relation corresponding to the owner entity type. Because of the existence dependence, the foreign key is declared is NOT NULL. The primary key of R is the combination of the partial key and the foreign key.

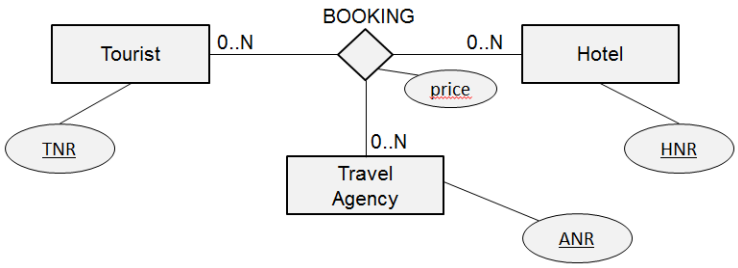


Room is a weak entity type which is existent dependent from hotel. We create **two relations**: HOTEL and ROOM. Room has a foreign key, HNR which is declared as NOT NULL and refers to hotel. Its primary key is the combination of RNR and HNR.

Relational model:
Hotel (HNR, Hname)
Room (RNR, *HNR*, beds)

To map a **n-ary relationship type**, we first create relations for each participating entity type. We then also define one additional relation R to represent the n-ary relationship type and add foreign keys to R referring to the primary keys of each of the relations corresponding to the participating entity

types. The primary key of R is a combination of all foreign keys. Attribute types of the n-ary relationship can be added to R.



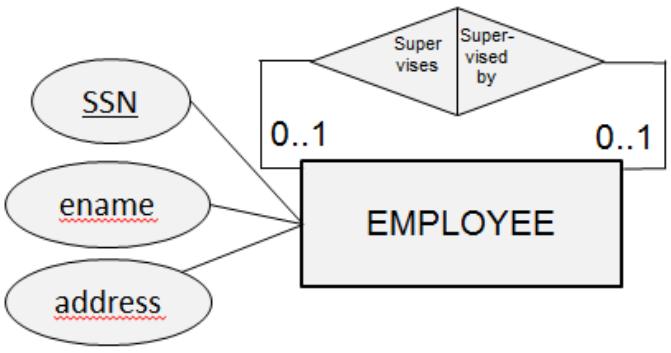
Booking is a ternary relationship type between tourist, hotel and travel agency. It has one attribute type: price. The relational model has relations for each of the three attribute types together with the relation BOOKING for the relationship type. The primary key of the latter is the combination of the three foreign keys. It also includes the price attribute. All six cardinalities are perfectly represented in the relational model.

Relational model:
TOURIST(TNR, …)
TRAV_AGENCY(ANR, …)
HOTEL(HNR, …)
BOOKING(*TNR, ANR, HNR*, price)

Unary or **recursive relationship** types can be mapped depending upon the cardinality.

- A recursive 1:1 or 1:N relationship type can be implemented by adding a foreign key referring to the primary key of the same relations.
- For a N:M relationship type, a new relation R needs to be created with 2 NOT NULL foreign keys referring to the original relation.

  ⇨ It is recommended to use role names to clarify the meaning of the foreign keys.



Add a foreign key supervisor the employee relation which refers to its primary key SSN. The foreign key can be null since it is possible that an employee is supervised by 0 other employees. Since the foreign key cannot be multivalued, an employee cannot be supervised by more than one other employee.

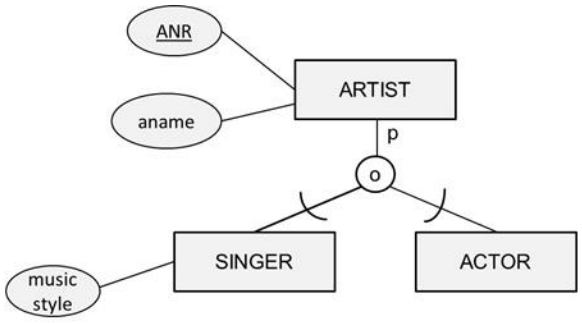
ER Model	Relational model
Entity type	Relation
Weak entity type	Foreign key
1:1 or 1:N relationship type	Foreign key
M:N relationship type	New relation with two foreign keys
N-ary relationship type	New relation with N foreign keys
Simple attribute	Attribute
Composite attribute	Component attributes
Multivalued attribute	Relation and foreign key
Key attribute	Primary or alternative key

A relational model is not a perfect mapping of the EER model. Some of the cardinalities have not been perfectly translated. Indeed, according to the multiple examples used in this section, it cannot be guaranteed that a department has at minimum one employee (not considering the manager). This is the case in the ER model. Besides, the same employee can be manager of multiple deparments. Some of the earlier major shortcomings of the EER model still apply here. We cannot guarantee that a manager of a department also works in the department. And we cannot also enforce that employees should work on projects assigned to departments to which the employees belong.

How could we map some of the EER concepts to the relational model?

- **Superclass/subclass relationships**

EER specializations can be mapped in various ways. The first option is to **create a relation for the superclass and each subclass** and link them with foreign keys. Another option is to **create a relation for each subclass** but not for the superclass. Finally we can create **one relation with all attributes** from the superclass and subclasses and add a type attribute.

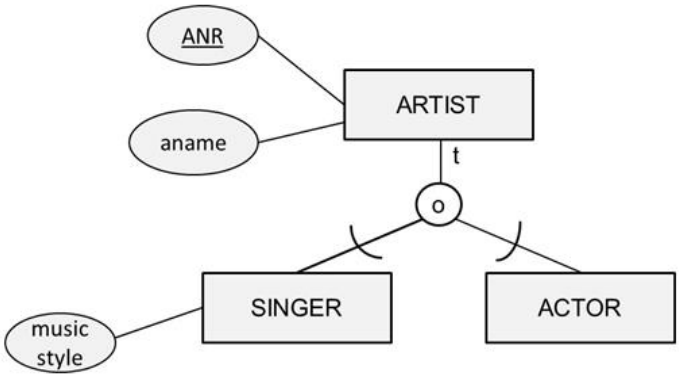


1st option: For this example, we will create three relations, one for the superclass (ARTIST) and two for the subclasses (SINGER and ACTOR). We add a foreign key ANR to each subclass relation which refers to the superclass relation. This foreign key also serves as primary keys.

/!\ In case de specialization would have been total and not partial, then we wouldn't have been able to enforce it with this solution. If the specialization would have been disjoint instead of overlapped, again, this wouldn't have been possible to model it with the first option.

2nd option: The second approach to map an EER model to a relational model is to only map **relations for the subclasses**. Let's illustrate this with a **total** and overlapping example. The attribute types of the superclass have to be added to the subclasses. The problem is that it creates redundancy. Therefore, this is not efficient from a storage perspective. Besides, this second option cannot enforce a relation to be disjoint, since the tuples in both relations can overlap.

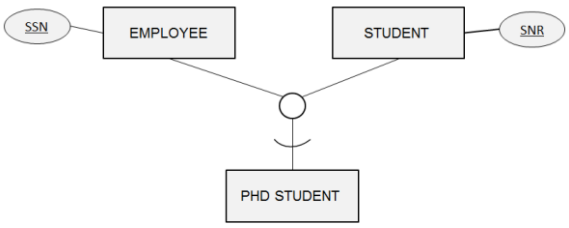
3rd option: Another option is to store all the subclass and superclass information in one relation. In this case, we add a type attribute "discipline" to indicate the subclass.



⇨ This approach can generate a **lot of NULL values** for the subclass specific attribute types.

- **Shared subclasses**

  In a specialization lattice, a subclass can have more than one superclass. Ex: a PHD student is both an employee as well as a student. This can be implemented in the relational model by **defining three relations**: EMPLOYEE, STUDENT and PHD. The primary key of the latter is a combination of two foreign keys referring to EMPLOYEE and STUDENT. This solution does not allow modeling a total specialization since we cannot enforce that all employees and student tuples are referenced in the PHD-student relation.

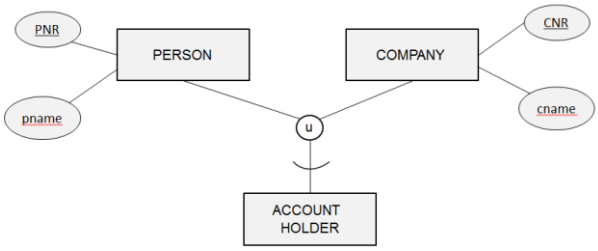


Relational model:
EMPLOYEE(SSN, …)
STUDENT(SNR, …)
PHD-STUDENT(*SSN,SNR*, …)

- **Categories**

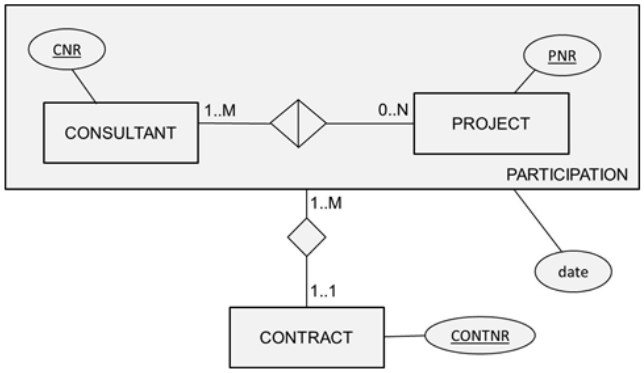
  Another extension is the concept of a **categorization**. In this case, the category subclass is a subset of the union of the entities of the super classes. Ex: an account holder can eider be a person or a company. This can be implemented in a relational model by creating a new relation that corresponds to the category and adding the corresponding attributes to it. We then define a new primary key attribute (a surrogate key) for the relation that corresponds to

the category. This surrogate key is then added as a foreign key to each relation corresponding to a superclass of the category. In case the super classes have to share the same key attribute types, then this one can be used in there is no need to add a surrogate key.
This solution is not perfect since we cannot guarantee that the tuples of the category relation are a subset of the union of the tuples of the super classes !



Relational model:
PERSON(PNR, …, *CustNo*)
COMPANY (CNR, …, *CustNo*)
ACCOUNT-HOLDER (CustNo, …)

- **Aggregations**



Aggregation is the third extension provided by the EER model.
In this example we aggregated the two entity type's consultant and project into an aggregation called participation. This aggregation has an attribute type date and participated in a 1 to M relationship type with the attribute type contract.

This can be implemented in the relational model by creating 4 relations: CONSULTANT, PROJECT, PARTICPATION, CONTRACT. The participation relation models the aggregation. Its primary key is the combination of two foreign keys referring to the consultant number and project number.

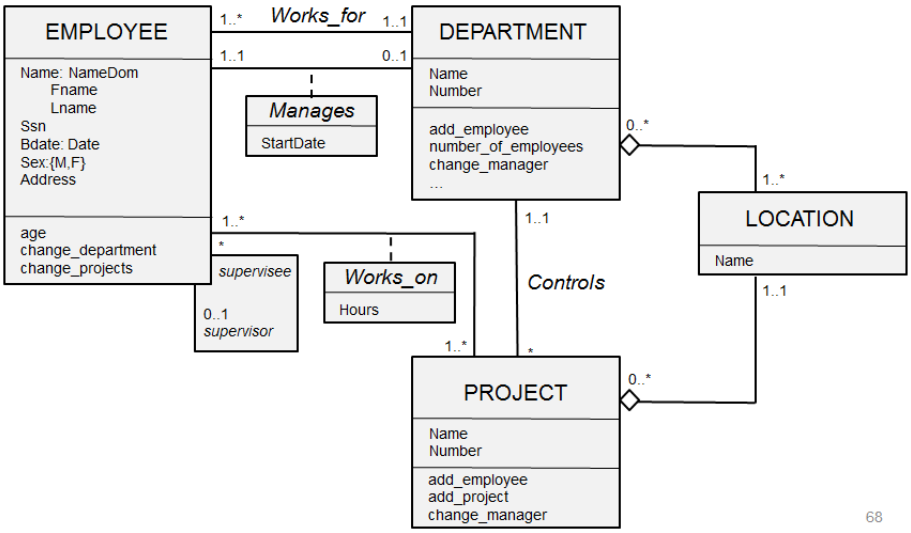
3.3 Mapping an EER model to an OO model

Mapping an EER conceptual model to a logical OO model is **unusual** since it would have been better to start off with a conceptual OO model. The OO model works with **classes** and **objects**. An object is an instance or occurrence of a class. Each class has both **attributes** and **methods**. Association types represent relationships between classes. Finally, inheritance and aggregation relationships between classes can be directly modeled.
The OO model offers **richer semantics** than the EER model and it would be therefore possible to further perfection the conceptual EER model.

EER Model	OO model
Entity type	Class
Simple attribute type	Attribute
Composite attribute type	Composite attribute
Multivalued attribute type	Separate class + Association type
1:1, 1:N, N:M relationship type	1:1, 1:N, N:M association type
N-ary relationship type	N-ary association type
Weak entity type	Qualified Association
Specialization/Categorisation/Aggregation	Specialization/Categorisation/Aggregation

In the OO logical model, methos can be added to the classes for further semantics refinements. A method has a method name, input and output argument, together with an implementation or body. This should all be carefully defined together with the business users.

Ex: method "Add-employee" wich allows to add an employee to the employee class. The input argument consists of all the employee data (SSN, date of birth, name, …). The output is void since this method does not return anything.



This model offers more semantics than the EER model.

3.4 Mapping an OO model to a CODASYL model

Mapping an OO model to a CODASYL model is about the **worst mapping** in terms of loss of semantics since we start from the riches conceptual model and implement it as the poorest conceptual model. Mapping like this could take place because of legacy environment.

OO model	CODASYL model
Class	Record type
Attribute	Data item
Component attribute	Data items
Multivalued attribute	Vector; repeated group
1:1, 1:N, N:M association type	Set type(s); dummy record type(s)
N-ary association type	Set type(s); dummy record type(s)
Qualified Association	Set type
Specialization/Categorisation/Aggregation	Set type(s); dummy record type(s)
Methods	-

⇨ Lots of loss of semantics ! It is important that it is documented and followed up with application code.

3.5 Mapping an OO model to a relational model

This is an approach which is frequently followed in the industry. Two approaches can be adopted here.

- **A pure relational approach:** maps an OO model to a **standard relational model**. Relations will be created and foreign keys defined to implement OO concepts such as associations, specialization, categorization and aggregation. Obviously this will involve a lot of semantics. Besides, methods cannot be implemented in a pure relational model.
- **Object relation approach**: extension of the relational model which add a selection of OO concepts such as named types, collection types, user-defined functions and inheritance.

OO model	Pure relational model
Class	Relation(s)/Foreign keys
Attribute	Attribute(s)/Relation(s)/Foreign keys
1:1, 1:N, N:M association type	Relation(s)/Foreign keys
N-ary association type	Relation(s)/Foreign keys
Qualified Association	Relation(s)/Foreign keys
Specialization/Categorisation/Aggregation	Relation(s)/Foreign keys
Method	-

3.6 Mapping an OO model to an OO model

This mapping is straight forward and may involve only some minor refinements of the model. No loss of semantics is incurred. The OO model can then be implemented in a transparent way using the OO DBMS. OO DBMS are not that popular in the industry due to their intrinsic complexity.

The mapping process can be supported by a case or a Computer Aided Software Engineering tool. This tool allows to automatically converting it to a logical data model. They also provide facilities to document any semantics that may get lost during the translation. The obtained logical data model

can be further tailored by making use of DBMS-specific modeling features and constructs to further improve performance or semantic richness.

4. Physical database design

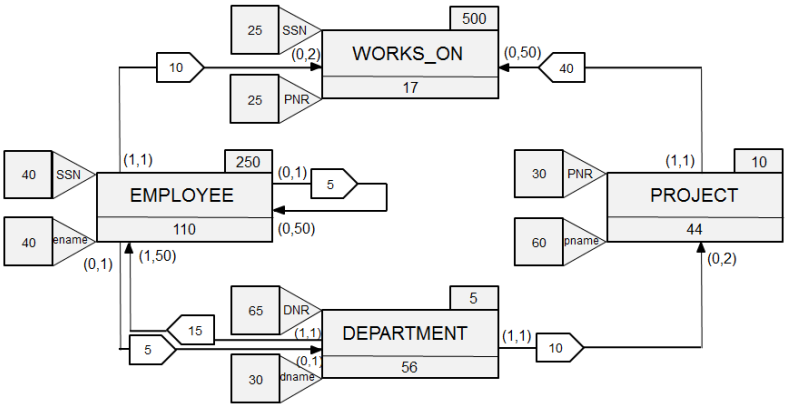
Once the logical database design has ended, we can continue with the physical database design. The goal of this step is to choose specific storage structures and access paths for the physical database files to achieve good performance for the various database applications. A key input for this step is the characteristics of the programs, transactions and queries which will interact with the database. Ideally, our database implementation should allow for physical data independence whereby changes to the physical design can be made without impact on the logical and external database model and applications.

3 key performance criteria :

- Response time is the time elapsed between submitting a database query and receiving a response. In a stock trading context, the response time is a critical parameter.
- Space utilization refers to the amount of storage space used by the database files and their access path structures on disk. In a big data setting, it should be carefully evaluated.
- Transaction throughput is the average number of transactions that can be processed per minute, per unit of time. Especially important for operational systems.

Another crucial piece of information is the **characteristics of the tables** and the **queries accessing them.**

- **Characteristics of tables:** amount and type of attributes, constraints, access rights, number of rows, estimated size and growth.
- **Characteristics of queries**: files that will be accessed by the query, attributes on which any selection conditions are specified, attributes on which any join conditions are specified, attributes whose values will be retrieved, frequency of execution, time constraints, amount of select/update/insert/delete queries, characteristics of select/update/insert/delete queries (attributes affected), etc.



Information displayed about the data size and usage. Rectangles represent the relations. We expect to have 250 employees, 5 departments, 10 projects and 500 works on tuples. Based upon the attributes, it is estimated that an employee tuple consumes 110 bytes of storage.

The anticipate access paths have been displayed. In 40% of the cases, employee information will be retrieved through either SSN or employee name, in 5% to the recursive entity type and in 50% through the department.

The average cardinalities have also been added: a department has on average between 1 and 50 employees. We can therefore estimate the storage space is needed and decide upon which access should be defined during this step.

During this step, it might be decided to **split up** some relations to **improve the performance.** Consider the relation EMPLOYEE (SSN, ename, streetaddress, city, sex, date of birth, photo, fingerprint, MNR, DNR).This relation contains quite a number of attributes. If it turns out that most applications and/or queries only retrieve the SSN and name of an employee than it might be possible to split the relation into two different relations EMPLOYEE1 (SSN, ename) and EMPLOYEE2(*SSN*, streetaddress, city, sex, date of birth, photo, signature, *MNR*, *DNR*). The first relation can then be used to serve the bulk of the application send to queries. In case all information needs to be retrieved, then both relations will have to be joined. Also denormalization might be considered to speed up the response time of database queries.
Suppose that many applications ask information about which employees work on what project. Each time, the number and name of employees asked for together with the number and name of the project. For a normalized database scheme, this would imply that the works on relationship needs to be joined with employee and project which is a resource intense operation. Hence, it might be considered to denormalize the data definitions by adding the employee name and project name attributes to the works on relation. This will allow for faster retrieval for requirements.
Consequence: inefficient use of storage and resulting anomalies.

In a distributed environment, it needs to be decided which data will be stored where and on what storage media. Tables need to be assigned to tables spaces which are then assigned to physical files stored on network resources. Related tuples can be physically clustered on an adjacent disk block to improve response time. Various types of indexing can also be added. An index represents a fast access path to the physical data. In a RDBMS environment, unique indexes will be automatically defined for the primary keys. Optional indexes can be designed for foreign keys or non-key columns that are often used as query selection criteria.
Cluster indexes can be created where the index order matches the physical order on disk.

A next step is to start preparing the data definition language or DDL statements for :

- Table definitions
- Definition of primary keys and declaration of integrity and uniqueness constraints
- Definition of foreign keys and declaration of referential integrity constraints
- Definition of the other columns and “NOT NULL" declarations
- Special integrity constraints: check constraints, triggers
- Definition of views
- Security privileges (read/write access)
- Output of this step are the DDL statements

The resulting DDL statements will then be compiled by the DDL compiler such that the data definitions can be stored in a catalog of the RDBMS. The outcome of this will be a database scheme and a collection of empty database files. The latter can then be loaded or populated with data. The database can now be accessed by application programmers by using data manipulation language (DML statements). The performance of the DML operations will be continuously monitored by

storing performance at this six such as response time, space utilization, transaction throughput raise in the catalog, etc. This will allow to carefully fine-tuning the performance of the database by optimizing indexes, database design and queries themselves.

CHAPTER 5 : Data languages in a relational environment

This chapter will present the languages in a relation data base environment. The SQL language may be considered one of the major reasons for the commercial success of relational databases. It indeed became a standard for relational databases; users were less concerned about migrating their database applications from other types of database systems. SQL is the most popular in use nowadays. This is one of the most database languages used in the industry!

The name SQL is presently expanded as Structured Query Language. Originally SQL was called SEQUEL.

1. Introduction to relational database systems

Relation database are based upon the relational model and managed by a **RDBMMS**. SQL is the language which is used for both **data dentition** and **data manipulation**. It is both **set oriented** and **declarative**. Indeed, as opposed to record oriented database languages, SQL can retrieve many records at a time. Furthermore, you only need to specify which data to retrieve in contrast to procedural database languages which also require to explicitly specifying the access path to the data. In addition, it has facilities for **defining views** on the database, for specifying **security** and **authorization**, for defining **integrity constraints**, and for specifying **transaction controls**.

Various versions have been introduced. Each relational database vendor provides its own implementation of SQL. Whereby typically the bulk of the standard is implemented, complemented with some vendor specific things, SQL can be used **interactively** and embe**d**ded.

SQL is the language of choice in the relational database environment. It can be used for both data definition as well as data manipulation. As a data definition language or DDL, it has a create table statement to define a logical scheme. *Create database*, *create table space* and *index statements* for the physical scheme and the *Create view* statement for the external scheme. SQL allows retrieving the data using a Select … From … statement. It is possible to add easily data by using the insert … Into function. It is also possible to use the Update or *Delete function*.

- **Free form language**: no specific notation is required as was the case for legacy programming language.
- Most SQL implementations are case intensive.
- Several commercial implementations exist, each providing their own character.

2. SQL Data Definition Languages

2.1 Schema and catalog concepts in SQL

SQL uses the terms table, row and column for the formal relational model terms relation, tuple, and attribute, respectively.
The key concept to start is the **SQL scheme**. This is a grouping of table and other database objects which logically belong together. An SQL scheme is identified by a **schema name** and it includes an **authorization identifier** to indicate the user or accounts who own the schema. They can perform any actions they want within the context of the scheme. A scheme is defined in the context of a business such as the order, the purchase order … Schema elements include tables, constraints, views, domains, and other constructs that describe the schema.

Ex : **CREATE SCHEMA** PURCHASE **AUTHORIZATION** 'Ysaline';

In general, not all users are authorized to create schemas and schema elements. The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

In addition to the concept of a schema, SQL uses the concept of a **catalog**, a named collection of schemas in an SQL environment.

2.2 The create table command in SQL

Once we have defined the schema we can start implementing SQL table. SQL tables implements a relation from the relational model. It specifies a new relation by giving it a name and specifies its attributes and initial constraints. It has multiple columns, one per attribute type and multiple rows, one for each tuple. An SQL table can be created by using the create table statements followed by the name of the table. It is recommended to explicitly assign a new table to an already existing scheme to avoid any confusions are inconsistencies.

We can explicitly attach the schema name to the relation name, separated by a period.
Ex: CREATE TABLE COMPANY.EMPLOYEE

CHAR(n)	Holds a fixed length string with size n
VARCHAR(n)	Holds a variable length string with maximum size n
SMALLINT	Small integer (no decimal) between -32768 to 32767
INT	Integer (no decimal) between -2147483648 to 2147483647
FLOAT(n,d)	Small number with a floating decimal point. The total maximum number of digits is n with a maximum of d digits to the right of the decimal point.
DOUBLE(n,d)	Large number with a floating decimal point. The total maximum number of digits is n with a maximum of d digits to the right of the decimal point.
DATE	Date in format YYYY-MM-DD
DATETIME	Date and time in format YYYY-MM-DD HH:MI:SS
TIME	Time in format HH:MI:SS
BOOLEAN	True or False
BLOB	Binary Large Object (e.g. image, audio, video)

An SQL table will have various colums, one per attribute table. Eache of these colums will have a corresponding **data type** to represent the format and the range of possible datavalues. Some examples are given in the table. These data types might be implemented differently in various different DBMSs.

2.3 Domains in SQL

It is also possible to define a user-defined databtype or domain in SQL. This can be handy when the domain can be reused multiple times in a table or in a scheme. Changes to the domain definitions then only have to occur once. It improves the **maintanability**.

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

2.4 Specifiying constraints in SQL

SQL column definitions can be further defined by imposing **colums constraints**.

- The **primary key** constraints define the primarey key of a table. It should have unique values and null values should be not allowed.
  Ex : Dnumber INT **PRIMARY KEY**;
- A **foreign key** constraint defines the foreign key which typically refers to the primary key of another, restricting the number of possible values.
- The **unique constraints** defines an alternative key of the table. It can also be specified directly for a secondary key if the secondary key is a single attribute.
- The **not null** constraint prohibits null values for a column. As SQL allows NULLs as attribute values, a constraint NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the primay key of each relation.
- The **default** constraint can be used to set a default value for a column. The default value is included in any new tuple if an explicit value is not provided for that attribute.if no default

clause is specified, the default “default value” is NULL for attributes that do not have the not null constraint.

- The **check** constraint can be used to define the constraint of a column value and restrict attribute or domain values.
  Ex : suppose that department numbers are resrricted to integer numbers between 1 and 20; then, we can cahnge the attribute declaration of Dnumber in the DEPARTMENT table.
  Dnumber INT NOT NULL CHECK (Dnumber > 0 AND Dnumber <21);

⇨ All the constraint shall be set in a close collaboration between the database developer and the business user.

Since many database objects are connected, we must specify how to manage the changes. This can be done by using referential **integrity constraints**.

- The on update cascade option says that an update should be cascaded to all referring objects.
- The under lead cascade options says the removal should be cascaded to all referring objects. If the option is set to restrict, the update a removal will be holed in case referring objects exist.
- Set null implies that all objects will be set null.
- Set default allows setting a default value.

The author statement can also be used to modify table columns communications. Common actions are: adding or dropping a column.

3. SQL Data Manipulation Language

From a DML perspective, SQL foresees four constructs:

- The select construct allows retrieving data from the relational database.
- The update and insert construct allows to modify and up date.
- The delete construct allow removing data.

3.1 SQL SELECT

By using the options we can ask specific and complex questions to the database. The SELECT construct allows retrieving date from the database. The result is a multi-set and not a set. In a set there are no duplicates and the elements are not ordered. In a multi-set the elements are not ordered but there can be duplicates elements in a multi-set.
SQL will not eliminate duplicates for many reasons:

- Duplicates elimination is an expensive operation.
- The user may also want to see duplicate tuples in a query result.
- Duplicates may also be considered by aggregate functions.

3.1.1 Simple queries

Simple queries are SQL statements that retrieve data from only one table. The FROM component in the SELECT instruction contains only one table name. The SELECT components then extract the columns required. It can contain many expressions which generally refer to the names of the columns we want to manipulate.

```
SELECT SUPNR, SUPNAME, SUPADDRESS, SUPCITY, SUPSTATUS
FROM SUPPLIER

          OR

SELECT * FROM SUPPLIER
```

SUPNR	SUPNAME	SUPADDRESS	SUPCITY	SUPSTATUS
21	Deliwines	240, Avenue of the Americas	New York	20
32	Best Wines	660, Market Street	San Francisco	90
37	Ad Fundum	82, Wacker Drive	Chicago	95
52	Spirits & co.	928, Strip	Las Vegas	NULL
68	The Wine Depot	132, Montgomery Street	San Francisco	10
69	Vinos del Mundo	4, Collins Avenue	Miami	92

It selects SUPNR, SUPNAME, SUPADDRESS, SUPCITY and SUPSTATUS **FROM** SUPPLIER. This query selects all information from the supplier table. So it provides a complete table doc. In case all columns of a table are requested, we can use a short cut: **SELECT*FROM** SUPPLIER.

It is also possible to select only a few columns (Ex : SUPNR and SUPNAME).

As mentioned before, the result of a SQL is a multi-set. There is an option which allows removing the duplicates from a multi-set: distinct select. **SELECT** DISTINCT SUPNR **FROM** PURCHASE_ORDER.

A missing WHERE clause indicates no condition on tuple selection. However, when a WHERE clause is added to the SQL statement, it specifies selection conditions to indicate which table rows could be selected. A number of operates could be used in the where clause such as comparison operators, Boolean, BETWEEN, IN, LIKE and NULL operator.

3.1.2 Queries with aggregate functions

Several expressions can be specified in the SELECT clause. This may also include **aggregate functions** which are used to summarize information from database tuples.
Ex : count, min, max, sum, average, standard deviation and variance.

3.1.3 Queries with GROUP BY/HAVING

Here, we want to apply the aggregate functions to **subgroups** of tuples in a table, where each subgroup consists of the tuples that have the same value for one or more columns. In other words, by using the GROUP BY clause, rows will be grouped when they have the same value for a specific column. The HAVING class can be added to retrieve the values of only groups of rows that **satisfy certain conditions**. It can only be used in a combination with a GROUP BY class and it can also include aggregate functions as the ones seen before.

3.1.4 Queries with ORDER BY

SQL allows the user to **order** the tuples in the result of a query by the values of one or more columns, using the ORDER BY clause. Each column, specified in the SELECT clause can be ordered and one can order on more than one column. The default is ascending. Depending upon the commercial implementations, null values may appear first or last in the sorting.

3.1.5 Join queries

These queries allow the user to join and combine data from different tables. The **FROM clause** in the select instruction specifies the **names** of the tables containing the rows we want to join. Additionally, it needs to be enforced under which **conditions** the rows of different tables can be joined. These conditions are specified in the **WHERE** clause. As with simple queries, it is possible to specify which columns from which tables we are interested in, and should be reported in the end result.

In SQL, the same names can be used for two or more columns as soon as the columns are in different tables. If this is the case and the FROM component refers to two or more columns with the same name, we must qualify the column name with the table name to prevent ambiguity. This is done by prefixing the table name to the column name.

An SQL join query without a WHERE component, corresponds to the Cartesian product of both tables (SELECT*FROM a,b). But this is not desirable … Join conditions can be specified in the WHERE component to ensure correct matches are made.

It is possible to join rows that refer to the same table twice. In this case, we should declare alternative table names, called **aliases**. The join condition is meant to join the table with itself by matching the tuples that satisfy the join condition.

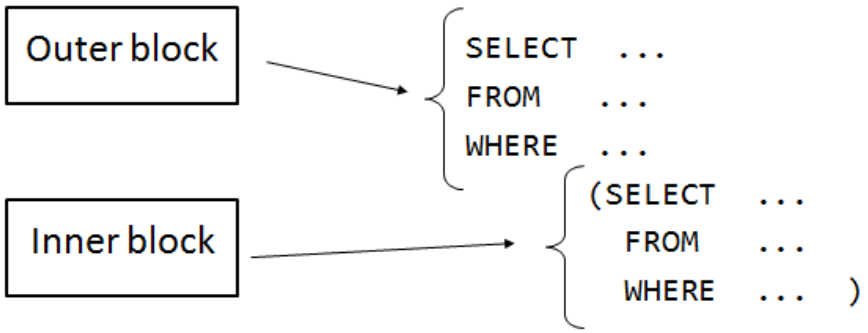
If we want to **eliminate duplicate** tuples from the result of a SQL query, we can use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result.

The join queries mentioned before are all **inner joins** which means that we only require an **exact match** before a tuple can be reported in the output. An **outer join** can be used when we want to keep all the tuples of one or both tables in the result of the JOIN, regardless of whether or not they have matching tuples in the other tables.

- **LEFT OUTER JOIN**: each row from the left table is kept in the result, if necessary completed with NULL values.
- **RIGHT OUTER JOIN**: each row from the right table is kept in the result, if necessary completed with NULL values.
- **FULL OUTER JOIN**: each row of both tables is kept in the result, if necessary completed with NULL values.

3.1.6 Nested queries

SQL queries can also be nested. Indeed, some queries require that existing values in the database be fetched and then used in a comparison condition. In other words, complete SELECT FROM blocks can appear in the work class of another query. The sub query or inner block is nested in the outer block. Multiple levels of nesting are allowed. The query optimizer will start by executing the query in the lowest block.



The WHERE component of an outer select block can contain an IN operater followed by a new (inner) select block.

3.1.7 Correlated Queries

In all previous cases, the nested sub-query (in the inner select block) could be entirely evaluated before processing the outer block. Nevertheless, this is not the case for **correlated nested queries**. Whenever a condition in the WHERE clause of a nested query references some column of a table declared in the outer query, the two queries are said to be correlated. The nested query is then evaluated once for each tuple (or combination of tuples) in the outer query.

The correlated queries allow implementing a looping mechanism by looping through the sub-query for each tuple of the table defined in the outer query block.

Correlated queries allow answering some complex questions

3.1.8 Queries with ALL/ANY

Besides the IN operator, the **ANY** and **ALL** operators can also be used to compare a single value to a multi-et.
The comparison condition ***v* > ALL *V*** returns TRUE if the value *v* is greater than ALL the values in the multi-set *V.* If the nested query doesn't return a value, it evaluates the condition as TRUE. The comparison condition *v > ANY V* returns TRUE if the value v is greater than at least one value in the multi-set V. If the nested query doesn't return a value, it evaluates the whole condition as FALSE. The comparison condition = ANY (…) is hence equivalent to the IN operator.

3.1.9 Queries with EXISTS

The EXISTS function in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. The result of EXISTS is a **Boolean value** TRUE or FALSE. In general, EXISTS **returns TRUE** if there is at least one tuple in the result of the nested query, and it returns **FALSE** otherwise. The NOT EXISTS operator returns TRUE if there are no tuples in the result of the nested query, and it returns FALSE otherwise. Because the EXISTS operator only evaluates whether or not a nested query outputs any rows, it is unimportant what is specified in the SELECT component. Hence, one commonly uses **SELECT*** for the sub-query.

3.1.10 Queries with sub-queries in SELECT/FROM

Besides the WHERE component, sub-queries can also appear in the SELECT or FROM component.

3.1.11 Queries with set operations

SQL also supports set operations to combine select blocks.

- **Union**: the result of this operation is a **table** that includes all tuples that are in **one of the SELECT blocks** or in both.
- **Intersect**: the result of this operation is a table that includes all tuples **that are in both SELECT blocks**.
- **Except**: the result of this operation is a table that includes all tuples that are in the **first SELECT block but not in the second**.

Duplicate tuples are by default eliminated from the result, unless the ALL operator is added: union ALL, intersect ALL, except ALL. Not all DBMS support these set operations.

3.2 SQL UPDATE

Modification to data can be made using the SQL UPDATE instruction.

3.3 SQL INSERT

The SQL INSERT allows adding data to a relational database. It is important to respect all constraints defined such as NOT NULL, integrity constraints, etc. when adding new tuples to the table.

3.4 SQL DELETE

Data can be removed using the SQL delete instruction. The remove of tuples should be carefully considered as it may impact other tables in a database. Referential integrity constraints keep the database in a consistent state in case of the removal of one or more reference tuples. The ON-delete cascade option cascades the delete operation to the referring tuples whereas the ON-delete restrict option prohibits removal of reference tuples.

4. SQL views

SQL views are part of the **external database scheme**. A view is defined by means of a SQL query and its content is generated upon invocation of the view by an application or other query. Hence it is a virtual table without physical tuples.
Views offer a number of advantages. Indeed, they facilitate the ease of use by hiding complex queries such as join queries or correlated queries from their users. Besides, they can also provide data protection by hiding columns or tuples from unauthorized users. Views are also a key component in a “three-layer” database architecture and allow for logical data independence. Views can be defined using the CREATE VIEW operator.
The database will modify queries that previews into queries on the underlying base tables.

Some views can be updated. In this case, the view serves as a window through which updates are propagated to the underlying base table(s). Updatable views require that INSERT, UPDATAE and DELETE instructions can be unambiguously mapped to INSERTs, UPDATEs and DELETEs on the underlying base table(s). If this property does not hold, the views I read only.

NB: A view update is feasible when only one possible update on the base tables can accomplish the desired update effect on the view.

Various requirements can be listed for views to be updatable. They depend upon the vendor of the arguments.

- No DISTINCT option in the SELECT-component
- No aggregate functions in the SELECT-component
- Only one table name in the FROM-component
- No correlated sub-query in the WHERE-component
- No GROUP BY in the WHERE-component
- No UNION, INTERSECT or EXCEPT in the WHERE-component

Another **issue** may arise in case an update on a view can be successfully performed. More specifically in case rows are inserted or updated through an updatable view, there is the chance that such rows no longer satisfy the view definition afterwards. In other words, the rows cannot be retrieved to the view anymore. The WITH CHECK option allows to avoid such "undesired" effects by checking the update and insert statements for conformity with the view definition.

5. SQL Indexes

Indexes are part of the physical database schema. An index provides a fast access path to the physical data in order to speed up the execution time of a query. It can be used to retrieve tuples with a specific column value in a quick way rather than having to read the entire table. Indexes can be defined over one or more columns. Syntax and implementations are RDBMS specific. They can be created in SQL using the CREATE INDEX statement. They can be removed using the DROP statement. The UNIQUE statement can be added to enforce that the values in the index are unique. Therefore, duplicated indexes would not be allowed. A CLUSTER index stores the tuples in a table based upon the index key. There can only be one CLUSTER index in a table as the tuples can only be stored in one order. If a table as no CLUSTER index, then the tuples will be stored in a random order.

6. SQL privileges

SQL also provides facilities to manage privileges. A privilege corresponds to the right to use certain SQL statements such as SELECT, INSERT, etc. on one or more database objects. Privileges can be granted or revoked. Both database administrator and schema owner can grant or revoke privileges to or from users or users' accounts. Privileges can be set at account, table or column level.

SELECT	Provides retrieval privilege
INSERT	Gives insert privilege
UPDATE	Gives update privilege
DELETE	Gives delete privilege
ALTER	Gives privilege to change the table definition
REFERENCES	Provides the privilege to reference the table when specifying integrity constraints.
ALL	Provides all privileges (DBMS specific)

7. Embedded SQL

SQL can be executed in both interactive as well as embedded mode. Many RDBMs users are never directly confronted with SQL constructs. They interact with applications which hide the complexity of SQL. These applications are written in general-purpose programming languages such as Java, C or COBOL combined with SQL statements. There are two broad options for using SQL statements in general purpose programming languages.

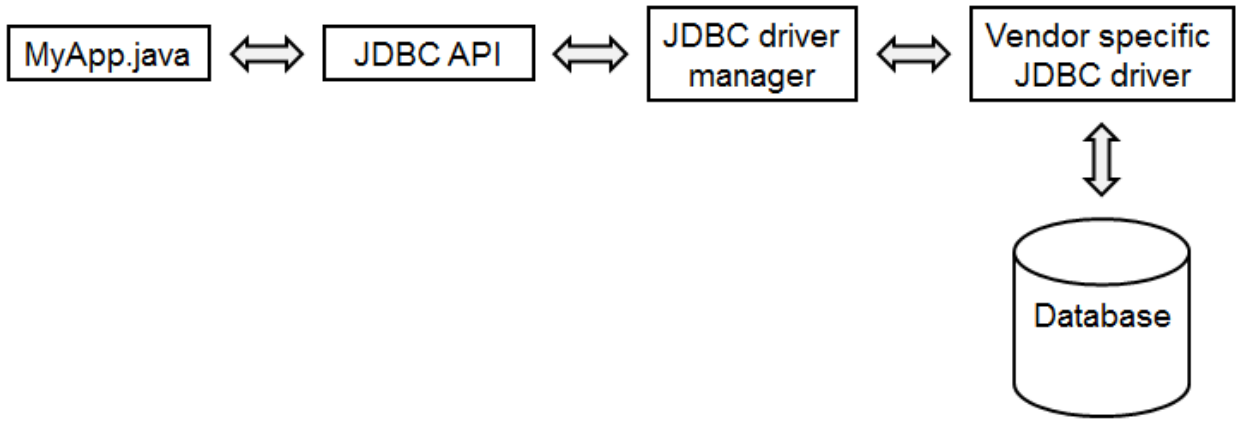
- **Statically embedded SQL**: the SQL Statements are directly embedded in a host language. Ex: SQLJ allows embedding a SQL statement in a Java host program.
- In dynamically embedded SQL, SQL statements passed as parameters to procedure calls from a call level interface. Ex: JDCB provides a set of procedures which allows the java program to communicate with the relational database.

⇨ To successfully execute SQL statement sin both settings, we need some additional languages constructs for the interaction between SQL and the host language.

A first important aspect concerns the **use of host language variables** in embedded SQL queries. A host language variable is a **variable declared in the host language**. It can be used in embedded SQL instructions to exchange data between SQL and the host language. A host variable used to pass database data to the application is called an output host variable. A host variable that is assigned a value by the application code, after which this value can be used in an SQL statement, is called an input host variable. Embedded SQL allows for host variables in any position where interactive SQL would allow for a constant.

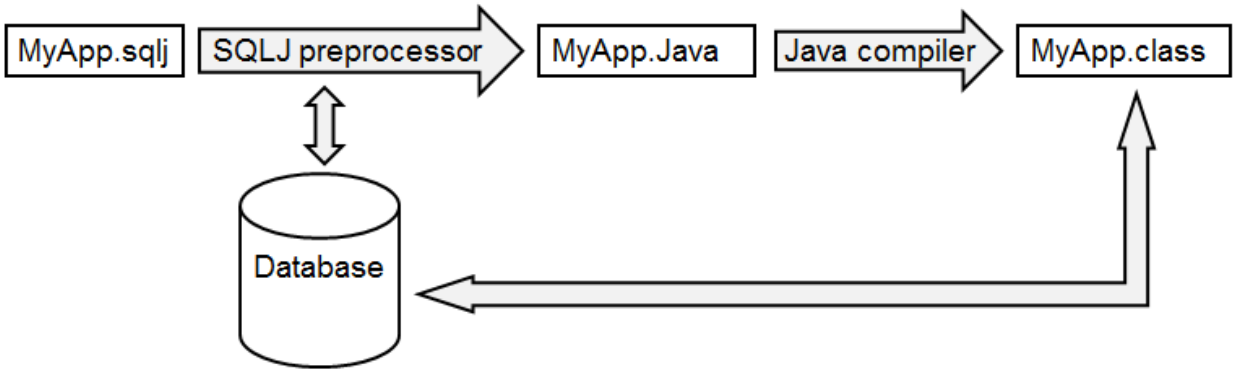
Because SQL is a set-oriented language, the query result will generally comprise **multiple tuples**. Host languages such as Java, C or COBOL are essentially record-oriented. Therefore, they cannot handle more than one record/tuple at a time. To overcome this impedance mismatch, the **cursor mechanisms** was introduced. Cursors allow for the tuples that result from an SQL query to be presented to the application code one by one. A cursor is to be declared and associated with a query. After that, the cursor allows to navigate through the tuples that make up the result of the query and offer them one by one to the application.

JDBC



The file MyApp.java contains the java source code of the database application. He uses the JDBC API to call the JDBC driver manager wich will use the vendor-specific JDBC driver to communicate to the unerlying relational database. A key advantage of JDBC is that it provides a uniform access interface to verious underlying data sources. Given its popularity, many database vendors provide support for it such as Microsoft, Oracle, IBM, etc. It is also supported by MySQL.

SQLJ



SQL statements can also be statically embedded in a host language. SQLJ is an example of this. It is built on top of JDBC and allows to embded SQL statemnts directly into Java programs. A preprocessor will then then transalate the SQL statement into Java or JDBC statements which are then further comilled by the java compiler to byte code. The preprocessor can perform syntax check, Java SQL type matchin and verifies the query against the database scheme at desired time. This will not only result in fewer errors at runtime but it also imroves the performance of the query.

SQL dilimiters separate the SQL instructions, to be processed by the precompiler from the native host language instructions. Each host language has its own SQL delimitor.

JAVA/SQLJ	#SQL {<sql instructions>};
C	EXEC SQL <sql instructions>;
COBOL	EXEC SQL <sql instructions> END-EXEC

To sum up, SQL is a very powerful database manipulation language. When using embedded SQL, it is highly recommended to push as much as possible complex data operations to the database instead of the application program. This will not only facilitate program maintenance but also improve the performance since every RDBMS has built sophisticated facilities for indexing and caching which have a huge impact on query execution time. Hence, it is of key importance that an application developer working with a DBMS, possesses a deep understanding of SQL in order to write high performing database applications.