Diplomatura en Big Data

Data Warehousing y OLAP

Alejandro VAISMAN
Departmento de Ingeniería Informática
Instituto Tecnológico de Buenos Aires
avaisman@itba.edu.ar

Quick Review of Database Concepts

Outline

- Database Design
- The Northwind Case Study
- Conceptual Database Design
- Logical Database Design
  - The Relational Model
  - Normalization
  - Relational Query Languages
- Physical Database Design

Steps in Database Design

- **Requirements specification**: Collects information about users' needs with respect to the database system
- **Conceptual design**: Builds a user-oriented representation of the database without any implementation considerations
  - Conceptual model: Entity-relationship is the most often used model; object-oriented model also applied, based on UML
  - Top-down design: Users' requirements merged before design begins, a unique schema is built
  - Bottom-up design: A separate schema for each group of users with different requirements, then, schemas merged in a global conceptual schema
- **Logical design**: Translates the conceptual schema to an implementation model, e.g., relational or object-relational
- **Physical design**: Customizes the logical schema from the previous phase to a particular platform, e.g., Oracle or SQL Server

Database Concepts

Outline

- Database Design
- **The Northwind Case Study**
- Conceptual Database Design
- Logical Database Design
  - The Relational Model
  - Normalization
  - Relational Query Languages
- Physical Database Design

Running Example: The Northwind Company

- The Northwind export company needs to design a relational database to manage the company data:
- **Customer** data: Identifier, name, contact person's name and title, full address, phone, and fax
- **Employee** data: Identifier, name, title, title of courtesy, birth date, hire date, address, home phone, extension, a photo (will be stored in the file system); employees report to higher level employees
- **Geographic** data: Territories where the company operates, organized into regions; an employee can be assigned to several territories, each territory can be linked to multiple employees
- **Shipper** data: Companies Northwind hires for product delivery; company name and phone number
- **Supplier** data: Company name, contact name and title, full address, phone, fax, and home page
- **Products** : Identifier, name, quantity per unit, unit price, an indication if the product has been discontinued, picture; each product has a unique supplier; products classified into categories
- **Sale orders**: Identifier, order date, due delivery date, actual delivery date, employee involved in the sale, customer, shipper in charge of delivery, freight cost, destination address
- An **order** can contain many products, for each of them the unit price, the quantity, and the discount that may be given must be kept

Database Concepts

Outline

- Database Design
- The Northwind Case Study
- **Conceptual Database Design**
- Logical Database Design
  - The Relational Model
  - Normalization
  - Relational Query Languages
- Physical Database Design

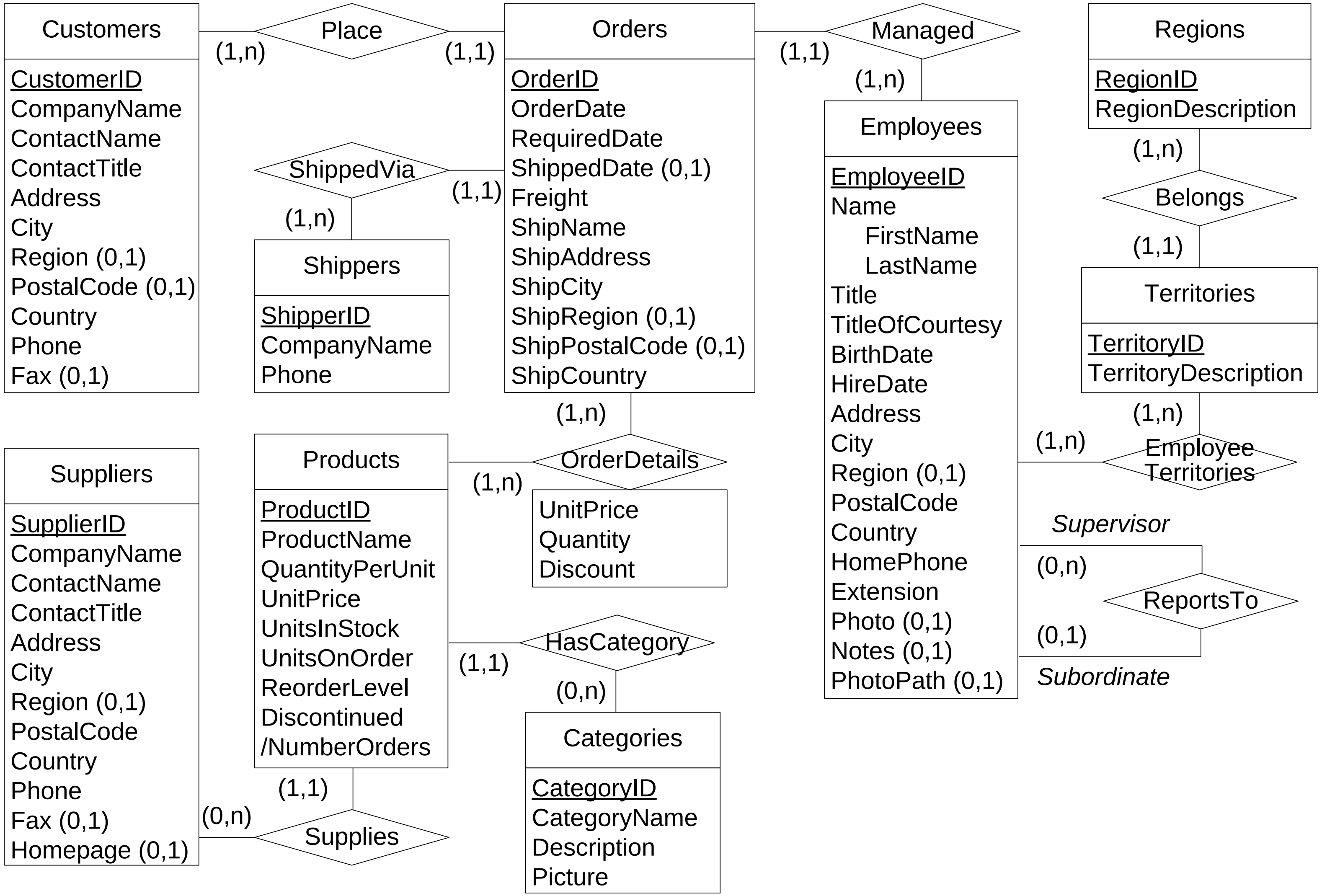
Entity-Relationship Model: Entity and Relationship Types

- Most often used conceptual model for database design
- **Entity type**: Represents a set of real-world objects of interest to an application
- **Entity**: Object belonging to an entity type
- **Relationship type**: Represents an association between objects
- **Relationship**: Association belonging to a relationship type
- **Role**: Participation of an entity type in a relationship type
- **Cardinalities in a role**: Minimum and maximum number of times that the entity may participate in the relationship type

Entity-Relationship Model: Entity and Relationship Types

- Types of roles
  - **Optional** vs. **mandatory**: minimum cardinality 0 vs. 1
  - **Monovalued** vs. **multivalued**: maximum cardinality 1 vs. n
- Relationship types
  - **Binary** vs. **n-ary**: two vs. more than two participating entity types
- Binary relationship types:
  - **One-to-one**, **one-to-many**, or **many-to-many** depending on the maximum cardinality of the two roles
- **Recursive** relationship types: the same entity type participates more than once in the relationship type
  - **Role names** are then necessary to distinguish different roles

Entity-Relationship Schema of the Northwind Database

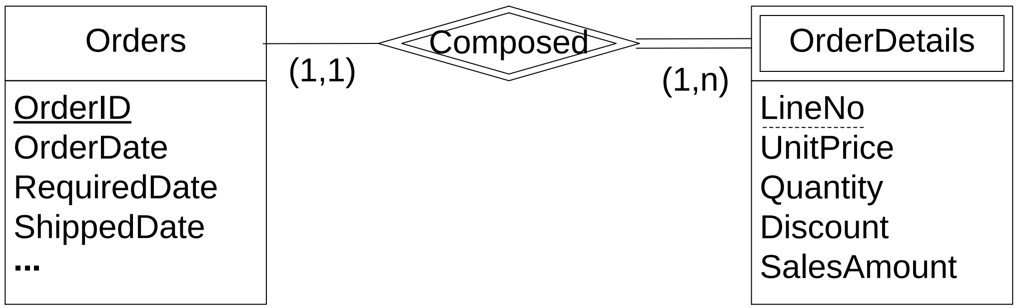


Entity-Relationship Model: Attributes

- **Attributes**: Structural characteristics describing objects and relationships
- Attributes have **cardinalities**, as for roles
- Depending on cardinalities, attributes may be
  - **Optional** vs. **mandatory**
  - **Monovalued** vs. **multivalued**
- **Complex** attributes: Attributes composed of other attributes
- **Derived** attributes: Their value for each instance may be calculated from other elements of the schema

Entity-Relationship Model: Identifiers and Weak Entity Types

- **Identifier** or **key**: One or several attributes that uniquely identify a particular object
  - **Regular entity types** also called strong entity types: have their own identifier
  - **Weak entity type**: do not have identifier of their own
- A weak entity type is dependent on the existence of another entity type, called the **identifying** or owner entity type
- An **identifying relationship type** relates a weak entity type to its owner
  - Normal relationship types are called **regular** relationship types
- **Partial key**: Set of attributes that can uniquely identify weak entities related to the same owner entity
- Relationship type OrderDetails modeled as a weak entity type

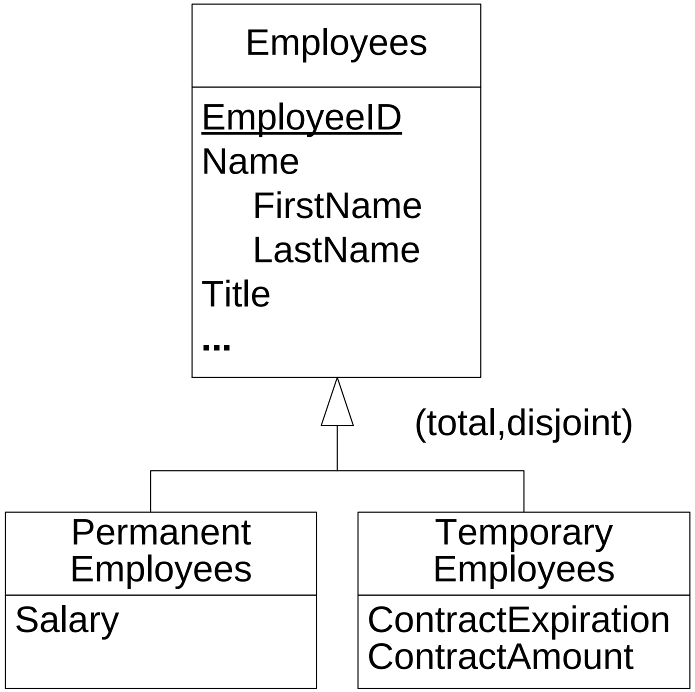


Entity-Relationship Model: Generalization

- **Generalization** (**is-a**) relationship: Perspectives of the same concept at different abstraction levels
  - Supertype: Entity at higher abstraction level
  - Subtype: Entity at lower abstraction level
- Population inclusion: Every instance of the subtype is also an instance of the supertype
- Inheritance: All characteristics (e.g., attributes, roles) of the supertype are inherited by the subtype
- Substitutability: Each time an instance of a supertype is required, an instance of the subtype can be used instead
- Types of generalizations
  - **Total** vs. **partial**: depending on whether every instance of the supertype is also an instance of one of the subtypes
  - **Disjoint** vs. **overlapping**: depending on whether an instance may belong to one or several subtypes
- **Multiple inheritance**: a subtype that has several supertypes
  - May induce conflicts when a property of the same name is inherited from two supertypes

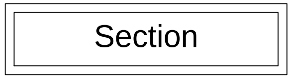
Entity-Relationship Model: Generalization

- Entity type Employees and two subtypes



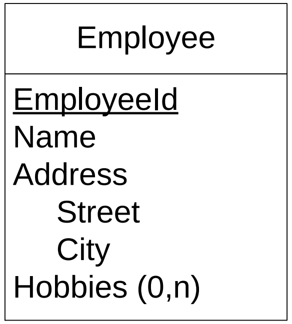
Entity-Relationship Model: Summary of Notation

Entity type
(short description)

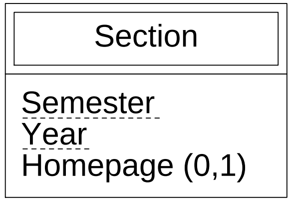


Weak entity type
(short description)

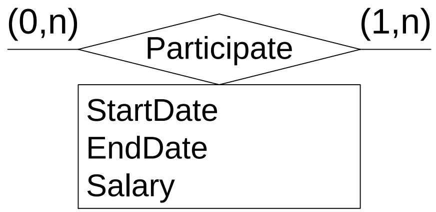
Relationship type
(short description)



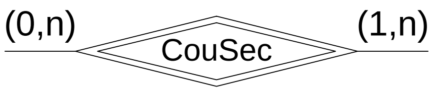
Entity type
(with attributes and key shown)



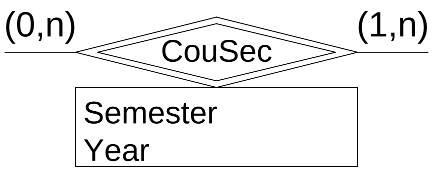
Weak entity type
(with attributes and partial key shown)



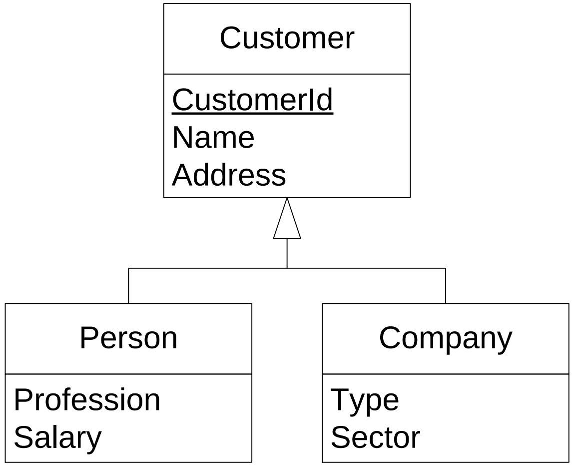
Relationship type
(with attributes shown)



Identifying relationship type
(short description)



Identifying relationship type
(with attributes shown)



Generalization relationship type

Database Concepts

Outline

- Database Design
- The Northwind Case Study
- Conceptual Database Design
- **Logical Database Design**
  - The Relational Model
  - Normalization
  - Relational Query Languages
- Physical Database Design

Relational Model

- Based on a simple data structure
  - A **relation** or **table**, composed of **attributes** or **columns**
  - Attributes must be **atomic** and **monovalued**
- Each attribute is defined over a **domain** or **data type**
  - Typical domains: Integer, float, date, string, . . .
- Provides several declarative **integrity constraints**
- **Tuple** or **row**: Element of the table
- **Not null attribute**: A value for the attribute must be provided
- **Key**: Column(s) that uniquely identify one row of the table
  - **Simple** vs. **composite** key: Depending on whether key is composed of one or several columns
  - **Primary** vs. **alternate** key: One of the keys must be chosen as primary by the designer, the others are alternate keys

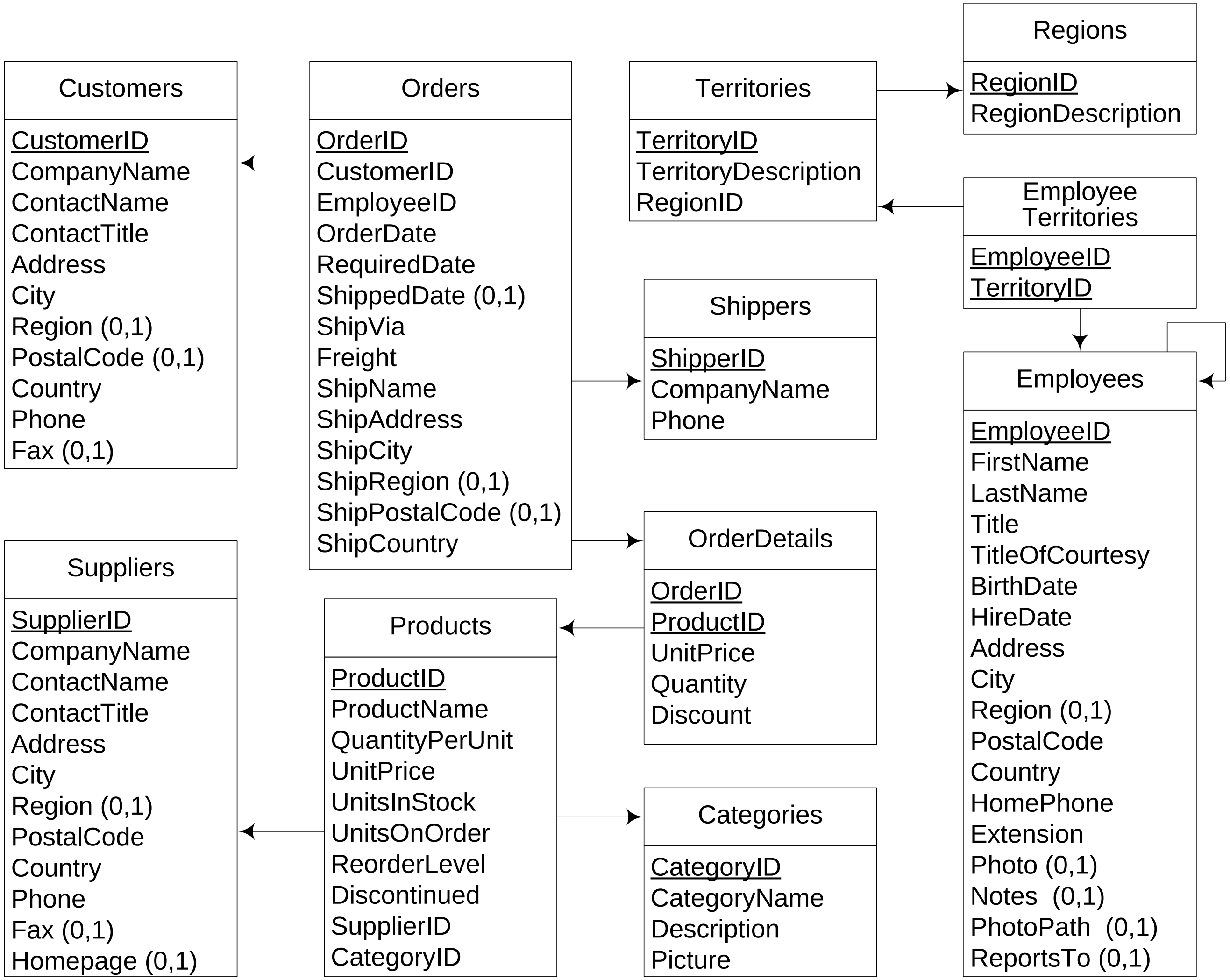
Relational Model

- Formally:
  - A relation R is defined by a **schema** $R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$
  - R is the name of the relation, A_i are attributes defined over the domain D_i
  - R is associated to a set of **tuples** (or **rows**) $r = \langle t_1, t_2, \ldots, t_n \rangle$
  - Set r is a subset of $D_1 \times D_2 \times \cdots \times D_n$, and it is called the **instance** or **extension** of R
  - The **degree** (or **arity**) of a relation is the number of attributes n in its schema R
  - The subset of the attributes that uniquely identifies a tuple is called a **key**, and we indicate it underlining the attribute name in the schema

Relational Model

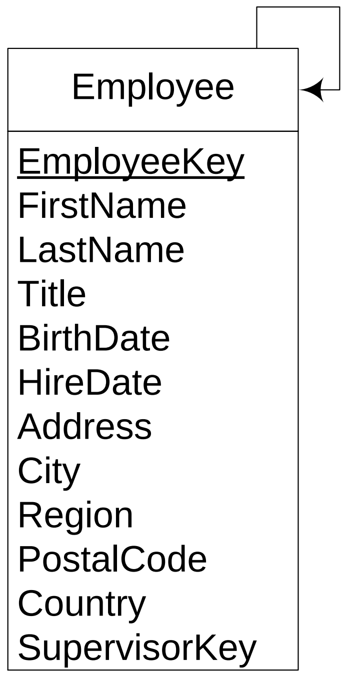
- **Referential integrity**
  - Defines a link between two tables (or twice the same one)
  - A set of attributes in one table (**foreign key**) references the primary key of the other table
  - Values in the foreign key columns must also exist in the primary key of the referenced table
- **Check constraint**: A predicate that must be valid when inserting or updating a tuple in a relation
  - Predicate can only involve the tuple being inserted/deleted
- All other constraints must be expressed by **triggers**: an event-condition-action rule that is automatically activated when a relation is updated

Relational Schema of the Northwind Database



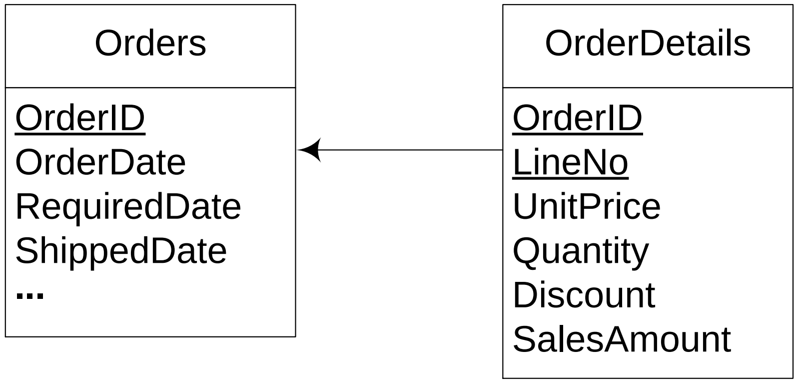
Translation from ER to Relational Schemas

- **Rule 1**: A **strong entity type** is associated with a table
  - Table contains the simple monovalued attributes and the simple component of the monovalued complex attributes of the entity type
  - Table also defines not null constraints for mandatory attributes
  - Identifier of the entity type defines the primary key of the table
- Example: Strong entity type Employees mapped to table Employees with key EmployeeID



Translation from ER to Relational Schemas

- **Rule 2**: A **weak entity type** is transformed as a strong entity type, but the table also contains the ID of the owner entity
  - Referential integrity constraint for this ID, to the table of with the owner entity type
  - Primary key: partial identifier of the weak entity type + identifier of the owner entity type
- Example: the weak entity type OrderDetails is mapped to a table of the same name
  - Key of the table: attributes OrderID (referencing table Orders) and LineNo



Translation from ER to Relational Schemas

- **Rule 3**: A regular **binary one-to-one relationship type** R between two entity types E_1 and E_2, mapped to tables T_1 and T_2, is mapped embedding the identifier of T_1 in T_2 as a foreign key
  - The simple monovalued attributes and the simple components of the monovalued complex attributes of R are included in T_2, not null constraints defined for the mandatory attributes
- Example: Suppose Supplies has cardinalities (1,1) with Products and (0,1) with Suppliers
  - Include SupplierID in table Products, as a foreign key, to avoid null values (there are more products than suppliers)

Products
ProductID
ProductName
QuantityPerUnit
UnitPrice
UnitsInStock
UnitsOnOrder
ReorderLevel
Discontinued
SupplierID
CategoryID

Translation from ER to Relational Schemas

- **Rule 4**: A regular **binary one-to-many relationship type** R relating entity types E_1 and E_2, where T_1 and T_2 are the corresponding tables, is mapped embedding the key of T_2 in table T_1 as a foreign key
  - The simple monovalued attributes and the simple components of the monovalued complex attributes of R are included in T_1, not null constraints for the mandatory attributes are defined
- Example: the one-to-many relationship type Supplies between Products and Suppliers is mapped by including SupplierID in table Products, as a foreign key

Products
ProductID
ProductName
QuantityPerUnit
UnitPrice
UnitsInStock
UnitsOnOrder
ReorderLevel
Discontinued
SupplierID
CategoryID

Translation from ER to Relational Schemas

- **Rule 5**: A regular **binary many-to-many relationship type** R between entity types E_1 and E_2, with mapping tables T_1 and T_2 is mapped as follows.
  - R is mapped to a table T containing the keys of T_1 and T_2, as foreign keys
  - The key of T is the union of these keys
  - Alternatively, the relationship identifier, if any, may define the key of the table
  - T also contains the simple monovalued attributes and the simple components of the monovalued complex attributes of R, and not null constraints for the mandatory attributes
- Example: the many-to-many relationship type EmployeeTerritories between Employees and Territories is mapped to the table Territories which contains the identifiers of the two tables involved

Employee Territories
<u>EmployeeID</u> <u>TerritoryID</u>

Translation from ER to Relational Schemas

- **Rule 6**: A **multivalued attribute** is associated with a table containing the identifier of the entity or relationship type to which it belongs
  - Corresponding referential integrity constraint is added
  - Primary key of table is composed of all its attributes
- Example: if attribute Phone of Customers is multivalued, it is mapped to a table CustomerPhone with attributes Customer and Phone composing the primary key

Customer Phone
CustomerID
Phone

Translation from ER to Relational Schemas

- **Rule 7**: a generalization between a supertype E_1 and subtype E_2 can be translated in three ways:
  - **Rule 7a**: Both E_1 and E_2 are mapped, respectively, to tables T_1 and T_2, the identifier of E_1 is propagated to T_2, and a referential integrity constraint relates this identifier to T_1
  - **Rule 7b**: Only E_1 is associated with a table T_1, which contains all attributes of E_2; attributes become optional in T_1
  - **Rule 7c**: Only E_2 is associated with a table T_2, i all attributes E_1 are inherited in T_2

Using Rule 7a

Employees
EmployeeID FirstName LastName ...

Permanent Employees
EmployeeID Salary

Temporary Employees
EmployeeID ContractExpiration ContractAmount

Using Rule 7b

Employees
EmployeeID FirstName LastName Title ... Salary (0,1) ContractExpiration (0,1) ContractAmount (0,1)

Using Rule 7c

Permanent Employees
EmployeeID FirstName LastName ... Salary

Temporary Employees
EmployeeID FirstName LastName Title ... ContractExpiration ContractAmount

Normalization

- In a relational schema, prevents potential **redundancies**, and **anomalies** in the presence of insertions, updates, and deletions
- Examples of relations that are not normalized

Order Details
OrderID
ProductID
UnitPrice
Quantity
Discount

Products
ProductID
ProductName
QuantityPerUnit
...
CategoryName
Description
Picture

Employee Territories
EmployeeID
TerritoryID
KindOfWork

- OrderDetails: each product associated with a discount percentage; discount information for a product p will be repeated for all orders in which p appears
- Products: category information (name, description, and picture) repeated for each product with the same category $\rightarrow$ potential inconsistency
- EmployeeTerritories: attribute KindOfWork has been added; if an employee can do many kinds of work, independently of the territories, information will be repeated as many times as the number of territories she is assigned to

Functional Dependencies

- The redundancies above characterized by the notions of **functional dependencies** and **normal forms**
- Given a relation R and two sets of attributes X and Y in R, a **functional dependency** $X \rightarrow Y$ (read X **determines** Y) holds if and only if, in all the tuples of the relation, each value of X is associated with at most one value of Y
- Note that a key determines all of the attributes in the relation
- F: a set of functional dependencies. Example:
  - in OrderDetails, ProductID $\rightarrow$ Discount holds;
  - in Products ProductID $\rightarrow$ CategoryID and CategoryID $\rightarrow$ CategoryName hold

Normal Forms

- Integrity constraints stating that a relational schema satisfies particular properties
- Relational model: only atomic and monovalued attributes; this restriction is called **first normal form**
- The **second normal form** avoids redundancies such as those in the table OrderDetails
- We need the concept of partial dependency:
  - Given a relation schema R, an attribute A in R is called a **prime attribute** if it belongs to some key in R. Otherwise it is called **nonprime**
  - Given a relation schema R such that X is a key of R, $Z \subset X$, and Y is a nonprime attribute in R, a dependency of the form $Z \rightarrow Y$ is called **partial**
- A relation R is in **second normal form (2NF)** with respect to a set of functional dependencies F no partial dependency is contained or can be derived from F.
  - Example: Product $\rightarrow$ Discount is a partial dependency, and the relation is not in 2NF
  - To make the relation comply with 2NF, the attribute Discount must be removed from OrderDetails and must be added to table Products

Normal Forms

- The **third normal form** avoids redundancies such as those in the table Products
- A dependency $X \rightarrow Z$ is **transitive** if there is a set of attributes Y such that the dependencies $X \rightarrow Y$ and $Y \rightarrow Z$ hold
- R is in **third normal form (3NF)** with respect to a set of functional dependencies F if it is in 2FN and no transitive dependencies between a key and a nonprime attribute exist in or can be derived from F.
  - Example: table Product does not satisfy 3NF, since there is a transitive dependency from ProductID $\rightarrow$ CategoryID and CategoryID $\rightarrow$ CategoryName
  - To comply with 3NF, the attributes dependent on CategoryID must be removed from the table, and a table Category must be defined
- The **Boyce-Codd normal form** avoids redundancies originated in functional dependencies
- A relation R is in **Boyce-Codd normal form (BCNF)** with respect to a set of functional dependencies F if, for every nontrivial dependency $X \rightarrow Y$ X is a key or contains a key of R
  - Example: All relations in the Northwind relational schema are in the Boyce-Codd normal form

Relational Query Languages: Structured Query Language (SQL)

- **Data definition language** (DDL) defines the database schema
- CREATE TABLE creates a table, defines attribute types, primary and foreign keys, and constraints
- DROP TABLE statement deletes a table, ALTER TABLE modifies the structure of a table

```
CREATE TABLE Orders (
        OrderID INTEGER PRIMARY KEY,
        CustomerID INTEGER NOT NULL,
        EmployeeID INTEGER NOT NULL,
        OrderDate DATE NOT NULL,
        RequiredDate DATE NOT NULL,
        ShippedDate DATE NOT NULL,
        ShippedVia INTEGER NOT NULL,
        Freight MONEY NOT NULL,
        ShipName CHARACTER VARYING (50) NOT NULL,
        ShipAddress CHARACTER VARYING (50) NOT NULL,
        ShipCity CHARACTER VARYING (50) NOT NULL,
        ShipRegion CHARACTER VARYING (50),
        ShipPostalCode CHARACTER VARYING (30),
        ShipCountry CHARACTER VARYING (50) NOT NULL,
        FOREIGN KEY CustomerID REFERENCES Customers(CustomerID),
        FOREIGN KEY ShippedVia REFERENCES Shippers(ShipperID),
        FOREIGN KEY EmployeeID REFERENCES Employees(EmployeeID) );
```

Relational Query Languages: Structured Query Language (SQL)

- **Data manipulation language** (DML) used to insert, update, and delete tuples from the database tables
  - Example: to add a new shipper in the Northwind database use INSERT statement
    ```
    INSERT INTO Shippers(CompanyName, Phone)
    VALUES ('Federal Express', '02 752 75 75')
    ```
  - To modify the tuple, use the UPDATE statement
    ```
    UPDATE Shippers
    SET    CompanyName='Fedex'
    WHERE  CompanyName='Federal Express'
    ```
  - To remove the new shipper use the DELETE statement
    ```
    DELETE FROM Shippers WHERE CompanyName='Fedex'
    ```

Relational Query Languages: Structured Query Language (SQL)

- Basic structure of an SQL expression to retrieve data:

  SELECT ⟨ list of attributes ⟩
  FROM ⟨ list of tables ⟩
  WHERE ⟨ condition ⟩

  - ⟨ list of attributes ⟩: attribute names whose values are to be retrieved by the query
  - ⟨ list of tables ⟩: relation names that will be included in the query
  - ⟨ condition ⟩ is a Boolean expression that must be satisfied by the tuples in the result
- Semantics of

  SELECT R.A, S.B
  FROM R, S
  WHERE R.B = S.A

  is given by:

  $\pi_{R.A,S.B}(\sigma_{R.B=S.A}(R \times S))$,
- SELECT : a projection π; WHERE: a selection σ; FROM: Cartesian product $\times$ between all tables
- An SQL query returns a set **with duplicates** (or a bag)

SQL: Selection, Projection, Set Operators

- Example: Identifier, first name, and last name of the employees hired between 1992 and 1994

```
SELECT EmployeeID, LastName
FROM   Employees
WHERE  HireDate >= '01/01/1992' and HireDate <= '31/12/1994'
```

- Union, intersection, difference, supported in SQL
  - Example: IDs of employees from the UK, or who are reported by an employee from the UK

```
SELECT EmployeeID
FROM   Employees
WHERE  Country='UK'
    UNION
SELECT ReportsTo
FROM   Employees
WHERE  Country='UK'
```

- UNION removes duplicates in the result, UNION ALL keeps them

The Join Operation

- We want the *name of the products supplied by suppliers from Brazil*
- The cartesian product combines data from the tables Products and Suppliers

ProductID	ProductName	SupplierID	SupID	Country
1	Chai	1	1	UK
2	Chang	1	1	UK
...	...	...	...	...
17	Alice Mutton	7	2	USA
18	Carnarvon Tigers	7	2	USA

- Last row combines a product supplied by supplier 7 with country of supplier 2, not useful → to filter out the meaningless tuples: a **Join** ($\bowtie$). In SQL:

```
SELECT ProductName, SupID
FROM   Suppliers S, Products P
WHERE  WHERE S.SupID = P.SupplierID AND S.Country = 'Brazil'
```

Or:

```
SELECT ProductName, SupID
FROM   Suppliers S JOIN Products P ON S.SupID = P.SupplierID
WHERE  WHERE S.Country= 'Brazil'
```

The Join Operation

- **Equijoin**: a join $R_1 \bowtie_\phi R_2$ such that ϕ is the equality between **all** the attributes with the same name in R_1 and R_2; projecting the result over the columns in $R_1 \cup R_2$ yields the **natural join** $R_1 * R_2$
- **Left outer join**: $R ⟕ S$, performs the join, but if a tuple in R does not satisfy the join condition the tuple is kept, and the attributes of S in the result are filled with null values
  - Example: Last name of employees and supervisor, or null if the employee has no manager
- Result:

EmployeeID	LastName	SupID	SupLastName
1	Davolio	2	Fuller
2	Fuller	NULL	NULL
3	Leverling	2	Fuller
...	...	...	...

The **Join** Operation

- The **right outer join** $R ⟖ S$, is analogous to the left outer join, except that the tuples that are kept are the ones in S
- The **full outer join** $R ⟗ S$, keeps all the tuples in both, R and S
  - Example: Last name of employees and supervisor, or null if the employee has no manager, and include employees who do not supervise anyone
    $\pi_{\text{EmployeeID,LastName,SupID,SupLastName}}(\text{Employees} ⟗_{\text{ReportsTo=SupID}} \text{Supervisors})$
- Result:

EmployeeID	LastName	SupID	SupLastName
1	Davolio	2	Fuller
2	Fuller	NULL	NULL
3	Leverling	2	Fuller
...	...	...	...
NULL	NULL	1	Davolio
NULL	NULL	3	Leverling
...	...	...	...

The Join Operation in SQL

- The join can be implemented as a projection of a selection over the Cartesian product of the relations
- The **join operation** is easier and more efficient to use
  - Example: Last name of employees, along with the last name of his supervisor, or NULL if the employee has no manager: LEFT OUTER JOIN operator

```
SELECT E.EmployeeID, E.LastName, S.EmployeeID, S.LastName
FROM   Employees E LEFT OUTER JOIN Employees S
       ON E.ReportsTo = S.EmployeeID
```

  - To include also the employees who do no supervise anybody, we use the FULL OUTER JOIN

```
SELECT E.EmployeeID, E.LastName, S.EmployeeID, S.LastName
FROM   Employees E FULL OUTER JOIN Employees S
       ON E.ReportsTo = S.EmployeeID
```

- Note that SQL is a **declarative language**, only have to tell the system **what** we want
- Relational algebra is an **operational language**, we must specify **how** we will obtain the result

SQL: Aggregation and Sorting

- Typically five basic aggregate functions: COUNT, SUM, MAX, MIN, and AVG.
- General form of an SQL query with aggregate functions

```
SELECT    ⟨ list of grouping attributes ⟩ ⟨ list of aggr_funct(attribute) ⟩
FROM      ⟨ list of tables ⟩
WHERE     ⟨ condition ⟩
GROUP BY  ⟨ list of grouping attributes ⟩
HAVING    ⟨ condition over groups ⟩
ORDER BY  ⟨ list of attributes ⟩
```

- The list of attributes in the SELECT clause must be the same as the one in the GROUP BY clause
- HAVING clause is analogous to the WHERE clause, applied over each group
- Result can be sorted with the ORDER BY clause, in ascendent or descendent order (ASC or DESC, respectively)

SQL: Aggregation and Sorting

- Example: Total number of orders handled by each employee, in descending order of number of orders; list only employees that handled more than 100 orders

```
SELECT    EmployeeID, COUNT(*) AS OrdersByEmployee
FROM      Orders
GROUP BY EmployeeID
HAVING    COUNT(*) > 100
ORDER BY COUNT(*) DESC
```

- Result

EmployeeID	OrdersByEmployee
4	156
3	127
1	123
8	104

- Example: For customers from Germany, total quantity of each product ordered; order the result by customer ID and product ID, in ascending order, and by quantity of product, in descending order

```
SELECT    C.CustomerID, D.ProductID, SUM(Quantity) AS TotalQty
FROM      Orders O JOIN Customers C ON O.CustomerID = C.CustomerID
          JOIN OrderDetails D ON O.OrderID = D.OrderID
WHERE     C.Country = 'Germany'
GROUP BY C.CustomerID, D.ProductID
ORDER BY C.CustomerID ASC, D.ProductID ASC, TotalQty DESC
```

SQL: Subqueries

- Used within a WHERE clause through two special predicates: IN and EXISTS (and their negated versions, NOT IN, and NOT EXISTS)
  - Example: Identifier and name of products ordered by customers from Germany

```
SELECT ProductID, ProductName
FROM   Products P
WHERE  P.ProductID IN (
         SELECT D.ProductID
         FROM   Orders O JOIN Customers C ON
                O.CustomerID = C.CustomerID JOIN
                OrderDetails D ON O.OrderID = D.OrderID
         WHERE  C.Country = 'Germany' )
```

  - Result

ProductID	ProductName
1	Chai
2	Chang
3	Aniseed Syrup
...	...

SQL: Subqueries

- Same query using the EXISTS predicate (yielding **correlated nested queries**)

```
SELECT ProductID, ProductName
FROM   Products P
WHERE  EXISTS (
       SELECT *
       FROM   Orders O JOIN Customers C ON
              O.CustomerID = C.CustomerID JOIN
              OrderDetails D ON O.OrderID = D.OrderID
       WHERE  C.Country = 'Germany' AND D.ProductID = P.ProductID )
```

- Result

ProductID	ProductName
1	Chai
2	Chang
3	Aniseed Syrup
...	...

SQL: Subqueries

- Use of the NOT EXISTS predicate
  - Example: Names of customers who have not purchased any product

```
SELECT C.CompanyName
FROM   Customers C
WHERE  NOT EXISTS (
       SELECT *
       FROM   Orders O
       WHERE C.CustomerID = O.CustomerID )
```

  - NOT EXISTS predicate will evaluate to true if, when P is instantiated in the inner query, the query returns the empty set
  - Result

CompanyName
FISSA Fabrica Inter. Salchichas S.A.
Paris spcialits

SQL: Views

- **View**: An SQL query stored in the database with an associated name
- Views are like virtual tables, that can be created from one or many tables or other views
- Views can be used to:
  - Allow data to be structured in a way that users find it natural or intuitive
  - Restrict access to data such that users can have access only to the data they need
  - Summarize data from various tables, which can be used for example to generate reports
- Created with the CREATE VIEW statement
- Once created, a view can be used in a query as any other table
- Example: Create a view ClientOrders that computes for each client and order the total order amount

```
CREATE VIEW ClientOrders AS (
        SELECT    O.CustomerID, O.OrderID,
                  SUM(D.Quantity*D.UnitPrice) AS Amount
        FROM      Orders O, OrderDetails D
        WHERE     O.OrderID = D.OrderID
        GROUP BY  O.CustomerID, O.OrderID )
```

SQL: Views

- View ClientOrders used in the next query to compute for each client the maximum amount among all its orders

```
SELECT     CustomerID, Max(Amount) as MaxAmount
FROM       ClientOrders
GROUP BY CustomerID
```

- Result

CustomerID	MaxAmount
ALFKI	1086.00
ANATR	514.40
ANTON	2156.50
AROUT	4675.00
BERGS	4210.50
...	...

SQL: Common Table Expressions

- A temporary table defined within an SQL statement
- Can be seen as a view within the scope of the statement
- Typically used when a user **does not have the necessary privileges** for creating a view
- Example:

```
WITH ClientOrders AS (
          SELECT    O.CustomerID, O.OrderID,
                    SUM(D.Quantity*D.UnitPrice) AS Amount
          FROM      Orders O, OrderDetails D
          WHERE     O.OrderID = D.OrderID
          GROUP BY  O.CustomerID, O.OrderID )
SELECT    CustomerID, Max(Amount) as MaxAmount
FROM      ClientOrders
GROUP BY  CustomerID
```

Database Concepts

Outline

- Database Design
- The Northwind Case Study
- Conceptual Database Design
- Logical Database Design
  - The Relational Model
  - Normalization
  - Relational Query Languages
- **Physical Database Design**

Physical Database Design

- Specifies how database **records** are **stored**, **accessed**, and **related** to ensure adequate **performance** of a database application
- Requires to know the specificities of an application, properties of data, and usage patterns
- Involves analyzing the transactions/queries that run frequently, that are critical, and the periods of time in which there will be a high demand on the database (**peak load**)

Performance of Database Applications

- Factors for measuring performance of database applications
  - **Transaction throughput**: # transaction processed in a time interval
  - **Response time**: Elapsed time for the completion of a transaction
  - **Disk storage**: Space required to store the database file
- A compromise has to be made among these factors
  - **Space-time trade-off**: Reduce time to perform an operation by using more space, and vice versa
  - **Query-update trade-off**: Access to data can be more efficient by imposing some structure upon it
- However the more elaborate the structure, the more time is needed to built it and maintain it when content changes
- After initial physical design is done, it is necessary to monitor it and tune it

Data Organization

- A database is organized in **secondary storage** into **files**, each composed of **records**, at their turn composed of **fields**
- In a disk, data is stored in **disk blocks** (**pages**)
  - Set by the operating system during disk formatting
- Transfer of data between main memory and disk takes place in units of disk blocks
- DBMSs store data on **database blocks** (**pages**)
- Selection of a database block depends on several issues
  - Locking granularity may be at the block level, not the record level
  - For disk efficiency, database block size must be equal to or be a multiple of the disk block size

File Organization

- Arrangement of data in a file into records and blocks
- **Heap files** (**unordered files**): Records placed in the file in the order as they are inserted
  - Efficient insertions, slow retrieval
- **Sequential files** (**ordered files**): Records **sorted** on the values of **ordering fields**
  - Fast retrieval, slow inserting and deleting
- **Hash files**: Use a **hash function** that calculates the **block** (**bucket**) of a record based on one or several fields
  - **Collision**: Bucket is filled and a new record must be inserted
  - Fastest retrieval, but collision management degrades performance

Indexes and Clustering

- **Indexes**: additional access structures that speed up retrieval of records in response to search conditions
- Provide alternative ways of accessing the records based on the **indexing fields** on which the index is constructed
- Many different types of indexes
  - **Clustering** vs. **nonclustering**: whether records in the file are physically ordered according to the fields of the index
  - **Single column** vs. **multiple column**: depend on the number of indexing files
  - **Unique** vs. **nonunique**: depend on whether duplicate values are allowed
  - **Sparse** or **dense**: whether there are index records for all search values
  - **Single-level** vs. **multilevel**: whether an index is split in several smaller indexes, with an index to these indexes
  - **Multi-level indexes** often implemented by using B-trees of B+-trees
- **Clustering**: Tables **physically stored together** as they share common columns (**cluster key**) and are often used together
  - Improves data retrieval
  - Cluster key stored only once $\Rightarrow$ storage efficiency