Relational Databases

(A) The Schema:

- An underlying data domain (data items/values as elements)

  $U = \{peter, mary, \$23, \$30, cd, book, cdWarehouse, chapters, john, ...\}$ (usually implicit, and possibly infinite)

- A binary (relational) predicate, *ParHood*, used to denote properties of two individuals at a time

  Its arguments are called "attributes", and usually have names: $ParHood(Parent, Child)$

- Attributes have (sub)domains, e.g. for *ParHood*:

  $Dom(Parent) = Dom(Child) = \{peter, mary, sue, stu, joe, \ldots\} \subseteq U$

- $Sales(Customer, Price, Article, Store)$, a 4-ary relational predicate (represents the structure of the table on page 24)

  A *name* for a property that applies to 4 individuals at a time

- Up to here, no data!

(B) The Relational Instance: D for (compatible with) the schema

- D is a structure with domain U
- With finite extensions for the predicates in the schema, i.e.

  For each n-ary predicate $P(A_1, \ldots, A_n)$ in the schema, a finite n-ary relation P^D

  That is, $P^D \subseteq Dom(A_1) \times \cdots \times Dom(A_n)$
- The extension of predicate *ParHood* is exactly the relation shown in the table on page 24

  The extension of predicate *Sales* is given by the table above, i.e. a finite set of 4-tuples:

  $$Sales^D \subseteq Dom(Customer) \times Dom(Price) \times Dom(Article) \times Dom(Store)$$

- Both relations are usual, classical, set-theoretic relations as seen in a discrete math course!
- The relational model can be provided in set-theoretic and logical terms

- Notice the separation between the relational schema and the relational database (the instance)
  - The schema does not have data, but it is metadata, i.e. data about the data

    In this case, how the data is organized and structured
  - The schema specifies the domain, relation names (database predicates), attributes (and other things ...)
  - The schema can be seen in some sense as the conceptual model of data
  - The extensions for the predicates in the schema provide, together, an instance for the schema
  - The database (instance) is said to be compliant (consistent) with the schema if it has the structure specified by the schema
  - Page 17 shows a database instance for the schema defined on page 26

- A RDB provides a clear and nice "logical view of data"
  - The user should be confronted with that logical view
  - Without having to care much about how the material, physical data is really stored in the computer
  - Nor about what internal data structures, access methods, or underlying algorithms for processing data
- This separation is crucial!

  An the big innovation proposed by E. Codd
- Relational databases are computationally represented and processed through relational database management systems (RDBMSs)

- Data are organized and represented in terms of relations
- Relations of fixed format (schema)

  Appropriate for representing highly "structured data"
- Data are processed via set-theoretic operations on relations

  Through the set-theoretic algebra of relations, a.k.a. Relational Algebra (RA)
- Languages of predicate logic, say Relational Calculus (RC), are used to specify relational predicates, schemas, constraints, queries, etc.

  A declarative language

  It expresses what we want, not how to achieve (compute) it
- In contrast, Relational Algebra is imperative

  We specify how to compute things

Queries

- Query: "Want the customers who have bough a CD"
- Relational algebra: $\Pi_{Customer}(\sigma_{Article = \text{'CD'}}(Sales))$

  Algebraic (operations with sets), imperative
- Intuitively (for now):

  - First, select from relation *Sales* all the tuples (rows) where attribute *Article* takes value *"CD"*

  - Next, project the result on attribute *Customer* (that is, forget the other attributes and their values)
- Relational calculus: $\exists y \exists z\ Sales(x, y, \text{'cd'}, z)$

  A logical formula, declarative
- Variable x is free (it is not quantified)

  Its possible values -when the formula becomes true in the instance- are the query answers

- $\underbrace{\exists y}_{\text{exists value for } y}\ \underbrace{\exists z}_{\text{exists value for } z}\ Sales(\underbrace{x}_{\text{free variable}}, y, \underbrace{\text{'cd'}}_{\text{a constant}}, \underbrace{z}_{\text{quantified variable}})$
- Query expresses what we want to retrieve, not how ...
- Variables y, z are existentially quantified

  They matter as long as there are values for them

  But we do not care about the values themselves ...

  Positions of (variables and constants in) relational predicate stay in correspondence with the relation schema, e.g. x in that position stand for an article
- RDBMSs have internal mechanisms for query evaluation
- As users we do not have to worry about them
- Well, not quite true ...

  Some internal operations may be costly, and it may be useful to know about that

- For an example, the previous query in RA can be expressed in a different, but equivalent form:

$$\Pi_{Customer}(\sigma_{Article = \text{'CD'}}(\Pi_{Customer, Article}(Sales)))$$

  If table *Sales* were too wide, it may be better to apply the projection first

- RDBMSs have internal query optimizers

  They can reformulate the query posed by the user, to reduce use of resources (space, time)

- A query posed by the user in SQL (usually close to RC), is reformulated into an optimized query in RA

- Extra Example (RC): "children of parents who bought a CD"

$$\underbrace{\exists x}_{\text{quantified now}}\ \exists y \exists z\ (Sales(\ \underbrace{x}_{\text{used to "join" the tables}}\ , y, \text{'cd'}, z)\ \underbrace{\wedge}_{\text{and}}\ ParHood(x, \underbrace{w}_{\text{the free var now}}\ ))$$

Example:

Supply	Company	Receiver	Item
	C	D_1	I_1
	D	D_2	I_2

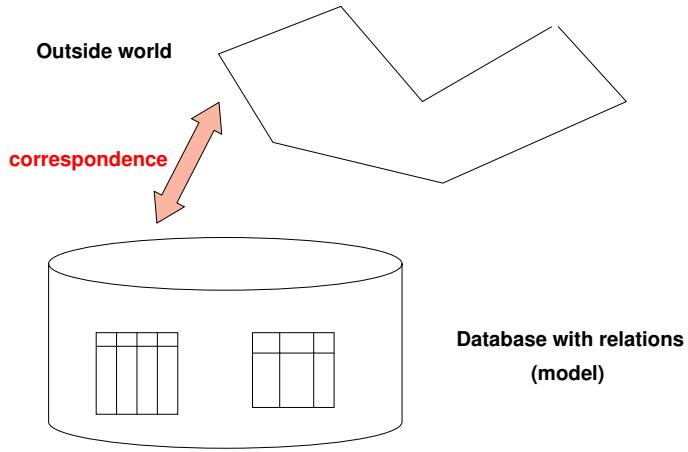
Articles	Item	Class
	I_1	K
	I_2	K

- A schema with an underlying domain, two relational predicates, of arities 3 and 2, resp.; and four attributes
- The extensions for the relational predicates are the relations shown in the tables
- Is this model capturing our outside reality?

  The "meaning" of the data as found in the application domain?
- If we understand that every item in relation *Supply* always belongs to a class in relation *Articles*, then our model is correctly reflecting this
- How to enforce that this is always the case (under updates)?

Relational Constraints

- We cannot emphasize enough: A database is a model of an external reality



- As a model it can be good or bad according to how well it represents and captures the external reality
- Integrity Constraints (ICs) help capture the meaning, the semantics, of data
- ICs (are intended to) keep the semantic correspondence between the world and the model of the world (the database)

- In the example, if we perform the update "*insert tuple* (C, D_3, I_4) *into Supply*", we obtain

Supply	Company	Receiver	Item
	C	D_1	I_1
	D	D_2	I_2
	C	D_3	I_4

Articles	Item	Class
	I_1	K
	I_2	K

  - This may not be admissible as a model of the real world
  - Not every supplied item is an official item ...
  - How can we prevent this from happening?

- The data model, i.e. the given relational schema, is not prohibiting this behavior
- We need more ...
- The previously mentioned ICs (a.k.a. consistency or semantic constraints)
- Conditions that instances of the schema should satisfy

- In this case we need an IC that is a referential IC:

  "*items in table Supply refer to items in table Articles*"

  Or better:

  "*every item appearing in table Supply appears in table Articles (assigned to some class)*"
- There are languages for expressing ICs as a part of the relational schema
- In the example, if this IC is a part of the schema and has to be satisfied, the update should not be accepted
- Usually (some kinds of) ICs become part of the schema
- Only some classes of ICs can be defined in SQL as a part of the schema (more later)

- Same example

Now with extensions:

Supply	Company	Receiver	Item
	C	D_1	I_1
	D	D_2	I_2

Articles	Item	Class
	I_1	K
	I_2	K
	I_2	H

- If in the outside world every item belongs to at most one class, this is not a correct model
- If we want "*every item belongs to at most one class*" to hold, it has to be stated as an IC, with the schema
- A cardinality constraint, namely a functional dependency:
  - *classes* are a function of the *items*, or, equivalently
  - *items* functionally determine the *classes*
- Notation: $\mathit{Articles}: \mathit{Item} \rightarrow \mathit{Class}$ (not logical implication)

Relational Models/Databases: Where From?

- The data model usually created before the DB is created
- Usually the design of a database starts with a conceptual model in ER form
  - More intuitive, closer to the outside reality
  - A high level model
  - Considering the participating classes (concepts, entities) to which data are associated
  - It is less of a model of the DB to come, but of the external reality
  - A conceptual *abstraction* that allows to understand, visualize, describe, ..., how data are organized
  - It describes the conceptual structure of the data stored in the DB: the concepts (classes, entities) and their relationships
  - This part does not involve the specific data items (values)

- Later, in the DB design phase, the conceptual model is transformed into a logical model, usually a relational model
- Some techniques are used to produce a set of relational predicates from the ER model
- A description of the relations (tables), etc., that will be created in the DBMS

  The relational schema emerges from the data model, before creating the DB
- A relational model can also be seen as a conceptual model

  But concepts and and relationships are rather implicit
- The schema is (represents) data of a different kind: data about data, i.e. metadata

  Metadata (schema, etc.) are stored in the DB and can be accessed (queried)

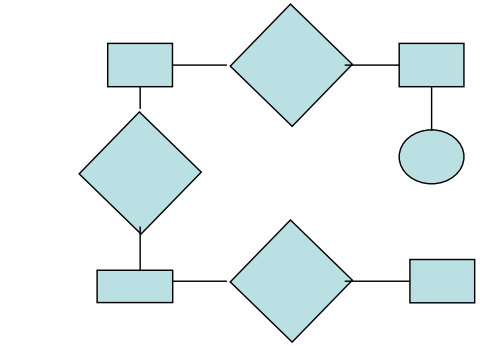
- The initial set of obtained relational predicates is "improved by additional transformations"
  - A new, right collections of tables and their logical connections
  - A normalization process via ICs
  - Avoiding, e.g. redundancy of data or updates anomalies
  - Obtaining a second set of tables
- Next, the resulting relational model is implemented in a RDBMS

  By creating the schema
- Finally, the database is populated (with data)

  Obtaining an instance
- Instances change frequently

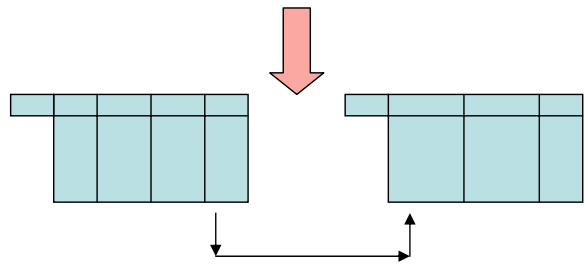
  Schemas not so much

  When they do, we have the problem of "schema evolution"



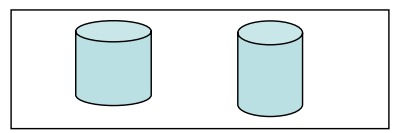
ER model (conceptual)

design



relational model (logical)

usually not a single step, involves "normalization" process



RDB

implementation

data appears here

RDBMS