

# **Brief Review of Relational Data Management and Databases**

(and a bit more)

Leopoldo Bertossi

School of Computer Science

# Data Models, ER, Relational Model

- To understand/manipulate/transform/update/extract/process ... data, we need to create the right **models of data**
- What is a model?
- An **abstraction**, a simplified **description or representation** of an external reality, phenomenon  
A physical phenomenon? A company? etc.
- The model explicitly captures some salient, relevant aspects of the external reality; others are left implicit
- We may be interested in extracting from the model both explicit and **implicit information**
- There is nothing like a universal modeling language
- If we want to represent mathematical relationships between numerical variables, we may use mathematical equations (they come in different flavors)

- If we want to model data, we may (and usually) create **mathematical models of data**
- In data management those models are directly based on **set theory and symbolic logic**
- Different ways of modeling data depending on the application and the (mathematical) elements we use to represent data
- A model of data has to capture the characteristics of data:
  - kinds of data items
  - associations/relationships between data items
  - associations between classes of data items
  - dependencies between data items
  - natural organization of data items (if any), etc. etc.

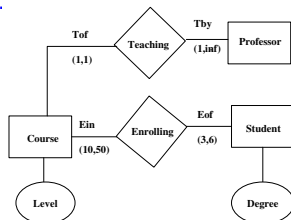
## ■ Example:

- *"Peter spends \$23 on a CD at CDWarehouse"*
- *"Mary spends \$30 on a book at Chapters"*
- Here "Peter" is a data item, the same applies to "\$23", "\$30", "book", "Chapters", ...
- Not only that: "Peter" is of the kind, say "Customer", "CD" of the kind "Article", ...
- We have **categories** (or **concepts**, **classes**, **entities**) of data items
- The concept "Customer" is related to the concept "Article"
- The data item "Peter" is related to data item "CD", but not to "book", etc.
- How can we capture all this?

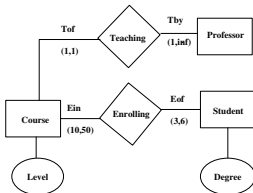
## ■ An Entity/Relationship (ER) Model:

A graphical “language” that uses **diagrams** to model data

- **Entities:** represented by boxes; they correspond to classes of data items of a same kind
- **Relationships:** by diamonds; they relate (data items from) different entities
- **Attributes:** by ovals hanging from entities; representing properties (of elements) of an entity  
May also hang from relationships
- **Cardinality constraints:** label links between relationships and entities



# Data Models, ER, Relational Model



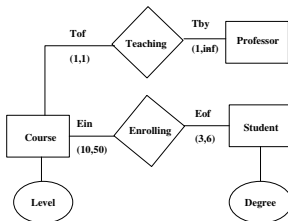
- Entities: *Course, Professor, Student*
- Relationships: *Teaching, Enrolling*
- *Course* represents courses, where students from ...
- *Students* are enrolled, and are taught by professors from ...
- *Professors*
- Attributes: *Degree, Level*

We could also have an attribute *Term* hanging from the *Enrolling* relationship

... an **attribute of the relationship** between students and courses

- *Course, Professor, Student* are also called “concepts” and the model above is also called a “conceptual model of data”
- An ER model is close to the outside (data) reality that is being modeled, close to how a user or modeler sees the world
- An ER model does not fully represent how data are represented, organized, structured, handled, ...
- ER is a high-level model that does not show much about the details of data
- An ER model, being a model, is expected to stay close to the outside reality that is being modeled
- That external reality gives a meaning (semantics) to the model

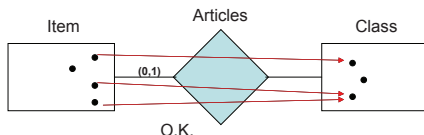
# Data Models, ER, Relational Model



- Cardinality constraints are used to capture more meaning  
For a better representation with the ER model of the external reality
- They are a kind of semantic constraints
- In the example, they capture:
  - The (external) limit on the number of students that may register in a course (between 10 and 50)
  - The limit on the number of courses that each student can take (between 3 and 6)
  - That each course is taught by exactly one professor, who has to teach at least one course



- Cardinality constrains can be understood as restrictions on the mappings between entities through the entity



- In particular, label  $(0,1)$  imposes that “at most one entity in *Class* is associated to each entity in *Item*”  
A very common constraint ...
- $(1,1)$  indicates “exactly one ...”
- $(1,N)$  indicates “at least one ...” (with  $N$  standing for a generic, arbitrary number)
- ER models can be extended by means of notation for indicating sub- or super-entities (i.e. subclasses, subconcepts, etc.), e.g. *GradCourse* can be defined as a subentity of *Course*
- With inheritance of relationships and attributes ...

- Relational Model of Data: (back to initial example, p. 4)
  - Notice that this data set is (seems to be) quite “structured”
  - What about this tabular representation?

Sales	Customer	Price	Article	Store
	peter	\$23	cd	cdWarehouse
	mary	\$30	book	chapters

ParHood	Parent	Child
	peter	mary
	john	stu

- Indeed a simplified representation
- It captures the relationships between data items through a same row in the table
- And the fact that data items are of different kinds
- Is this a mathematical model?
- It can be the tabular presentation of a mathematical model
- Based on set-theory and predicate logic, consisting of:

## (A) The Schema:

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

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

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

Its arguments have names called “attributes”, denoted *ParHood*(*Parent*, *Child*)

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

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

- *Sales*(*Customer*, *Price*, *Article*, *Store*), a 4-ary **relational predicate** (representing the structure of the table above)

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  
More precisely, a **finite**  $n$ -ary **relation**  $P^D$  for each  $n$ -ary predicate  $P(A_1, \dots, A_n)$  in the schema  
That is,  $P^D \subseteq \text{Dom}(A_1) \times \dots \times \text{Dom}(A_n)$
- In the example, the extension of predicate *ParHood* is exactly the relation given by the table  
The extension of predicate *Sales* is given by the table above, i.e. a finite set of 4-tuples:  
$$\text{Sales}^D \subseteq \text{Dom}(\text{Customer}) \times \text{Dom}(\text{Price}) \times \text{Dom}(\text{Article}) \times \text{Dom}(\text{Store})$$
- 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** itself
  - 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 with the schema** if it has the structure specified by the schema
  - Page 4 shows a database instance for the schema defined in page 11

# Relational Databases

- The relational database 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
  - Or by what means data structures and access methods are used
- Relational databases are computationally represented and processed through relational database management systems
  - 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, aka. relational algebra
  - Languages of predicate logic, say relational calculus, are used to express relational predicates, schemas, constraints, queries, etc.

Query: “Want the customers who have bought a CD”

- Relational algebra:  $\Pi_{Customer} \sigma_{Article = 'CD'}(Sales)$

Algebraic, imperative

- Relational calculus:  $\exists y \exists z Sales(x, y, 'cd', z)$

Logical, declarative

Variable  $x$  is not quantified (it's free), and its possible values are the answers

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 stay in correspondence with the relation schema

(More examples coming ...)

## Example:

<i>Supply</i>	Company	Receiver	Item
	<i>C</i>	<i>D</i> <sub>1</sub>	<i>I</i> <sub>1</sub>
	<i>D</i>	<i>D</i> <sub>2</sub>	<i>I</i> <sub>2</sub>

<i>Articles</i>	Item	Class
	<i>I</i> <sub>1</sub>	<i>K</i>
	<i>I</i> <sub>2</sub>	<i>K</i>

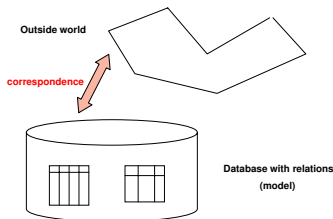
- A **schema** with an underlying domain, two relational predicates, of arities 3 and 2, resp.; and four attributes
- The **extensions** for the relational predicates are the relations shown in the tables
- Is this model capturing our outside reality?

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



- 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 it represents and captures the external reality
- 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

<i>Supply</i>	Company	Receiver	Item
	<i>C</i>	<i>D<sub>1</sub></i>	<i>I<sub>1</sub></i>
	<i>D</i>	<i>D<sub>2</sub></i>	<i>I<sub>2</sub></i>
	<i>C</i>	<i>D<sub>3</sub></i>	<i>I<sub>4</sub></i>

<i>Articles</i>	Item	Class
	<i>I<sub>1</sub></i>	<i>K</i>
	<i>I<sub>2</sub></i>	<i>K</i>

- 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 ...**

# Relational Constraints

- We have add **integrity constraints** (ICs) (aka. 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**

# Relational Constraints

- Same example, but now with the extensions

<i>Supply</i>	Company	Receiver	Item
	<i>C</i>	<i>D<sub>1</sub></i>	<i>I<sub>1</sub></i>
	<i>D</i>	<i>D<sub>2</sub></i>	<i>I<sub>2</sub></i>

<i>Articles</i>	Item	Class
	<i>I<sub>1</sub></i>	<i>K</i>
	<i>I<sub>2</sub></i>	<i>K</i>
	<i>I<sub>2</sub></i>	<i>H</i>

- 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 form of **cardinality constraint**, namely a **functional dependency**:
  - *classes* are a function of the *items*, or, equivalently
  - *items* functionally determine the *classes*

**Notation:** *Articles*:  $Item \rightarrow Class$  (not logical implication)

- The **data model** usually created before the DB is created
- Usually **design of a database** starts with a conceptual model in ER form
  - More intuitive, closer to the outside reality, and in more general terms
  - Considering the participating elements in it to which data is 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, raw data

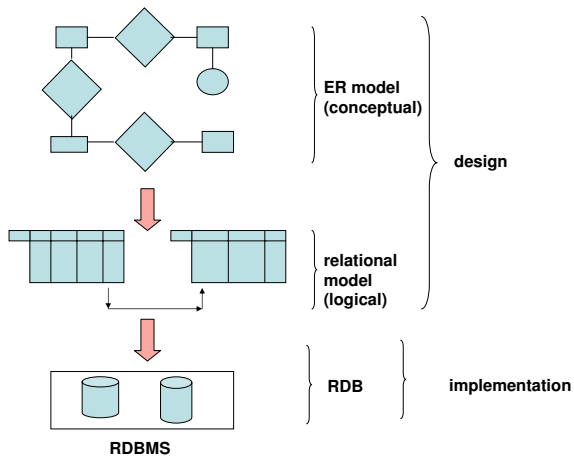
- Later, in the DB design phase, **the conceptual model is transformed into a logical model**, usually a relational model
- Some techniques are used to produce a set of relational predicates from the ER model
- A description of the relations (tables), etc., that will be created in the DBMS

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 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”

# Relational Models





- The relational model of data provides a **declarative query language**
- It allows us to tell the DB **what** data we want, without having to specify **how** to get it
- A simple query language is based on predicate logic
- A logic-based query language for relational databases is called the **relational calculus**
- From the example DB in page 18, we want to obtain the classes of articles that are provided by company *D*

Supply	Company	Receiver	Item
	C	D <sub>1</sub>	I <sub>1</sub>
	D	D <sub>2</sub>	I <sub>2</sub>
	C	D <sub>3</sub>	I <sub>4</sub>

Articles	Item	Class
	I <sub>1</sub>	K
	I <sub>2</sub>	K



Answer	Class
	K

- A **query is a mapping** that sends instances to single-table instances, with a possibly different schema



- Is there a language to specify (write) the query?
- Something that can be processed by the DBMS?
- Different languages ...
- General issue: Given a specific query, can it be captured by (expressed in) a given query language?
- The query above can be expressed as a **formula of predicate logic**
- In DBs, it takes the form of a **relational calculus** query:

$$\exists y \exists z (\text{Supply}(D, y, z) \wedge \text{Articles}(z, x))$$

- $\exists y \exists z (Supply(D, y, z) \wedge Articles(z, x))$  (\*)
  - This is a query  $Q(x)$ , with single **free variable**,  $x$  (variables  $y, z$  are **bound** due to the existential quantifiers)
  - The values for  $x$  that make the condition (expressed by the query) true on the given instance are the answers to the query
  - The other variables are existentially quantified  
Their specific values do not matter as long as they exist (and satisfy the condition of the query)
  - The double occurrence of  $z$  captures the fact that we combine the two tables through values in common for items
  - The answers are the values that can take the variable  $x$  when the the formula is true in the database
- $K$  is an answer, because (\*) is true in the instance: **there is** an item value for  $z$ , e.g.  $I_2$ , and **there is** a receiver value for  $y$ , e.g.  $D_2$ , such that

$$Supply(D, D_2, I_2) \wedge Articles(I_2, \underline{K})$$

becomes true in the instance

- A completely **declarative** query!
- Also **symbolic**, follows a precise syntax (grammar), and machine processable!
- The relational model also offers imperative, algebraic, set-theoretic query language: the **relational algebra** (RA)
- The same query now in RA:

$$\Pi_{Class} \sigma_{Company='D'} (Supply \bowtie_{Item} Article)$$

- RA and relational calculus are provably equally expressive
- Both of them are the basis for the common language offered by RDBMSs: **SQL** (Structured Query Language)
- As an SQL query:  
SELECT Class  
FROM Supply, Article  
WHERE Supply.Company = 'D' AND Supply.Item = Article.Item

- Another SQL query:

```
SELECT Name  
FROM   Accounts  
WHERE  Balance > 10,000
```

asking for the values of attribute *Name* in relation *Accounts* of those customers who have a balance greater than 10,000

- In relational calculus (predicate logic):

$$Q'(\mathbf{x}) : \exists u \exists z (Accounts(u, \mathbf{x}, z) \wedge z > 10,000)$$

- SQL can also be used to create, modify and access metadata, e.g. to retrieve elements of the schema

- A RDBMS is able to take a declarative query (in SQL) and develop an internal **query evaluation plan**:

Which tables to access?, When?, How?, Order?,  
Combining partial results? Which ones? How? ...

- **Query evaluation can be optimized**:
  - By making use of statistics, indices, etc.
  - **Syntactic query optimization**: syntactically rearrange the query making it easier to compute  
E.g. if possible, better apply a selection before a join, because the join -an expensive operation- is reduced
  - **Semantic query optimization**: Take advantage of explicit ICs to optimize query answering
    - If we want the addresses of a given client, and the former functionally depend upon the latter, return the first address found (no need to search for more)
    - If an IC says managers make at least 100K, a query asking for managers making less than 80K can get empty answer w/o checking the table

- The logic-based language (say, relational calculus) can also be used to state ICs

E.g. functional dependencies (FDs):

*“items cannot be associated with more than one class”*

$$\forall x \forall y \forall z (Articles(x, y) \wedge Articles(x, z) \rightarrow y = z)$$

This is a **sentence**, i.e. a formula without free variables


A declarative IC!      Again, symbolic!

Evaluated as a query in a consistent instance, the answer should be: Yes!

# Integrity Constraints (revisited)

- Example: As before

<i>Supply</i>	Company	Receiver	Item
	<i>C</i>	<i>D</i> <sub>1</sub>	<i>I</i> <sub>1</sub>
	<i>D</i>	<i>D</i> <sub>2</sub>	<i>I</i> <sub>2</sub>



<i>Articles</i>	<u>Item</u>	Class
	<i>I</i> <sub>1</sub>	<i>K</i>
	<i>I</i> <sub>2</sub>	<i>K</i>

- We can express the *referential IC* from *Supply* to *Articles*:

*“every item value in the former must appear in the latter  
(the official list of items)”*

$$\forall x \forall y \forall z (Supply(x, y, z) \rightarrow \exists w Articles(z, w))$$

- We can also impose the condition that *Item* is a **key** for relation *Articles*  
I.e. all attributes of *Articles* functionally depend upon *Item*  
(*Articles*: *Item*  $\rightarrow$  *Class*)
- The combination of the two is a **foreign key constraint** on *Supply*: Its attribute *Item* is the key in a foreign relation



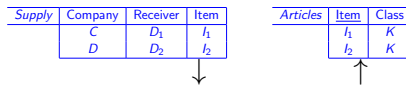
# Integrity Constraints (revisited)

- **Database maintenance** is the problem of keeping an instance **consistent**

I.e. satisfying the specified ICs when it undergoes updates

Many issues around this problem ...

Example: As above, with **foreign key constraint** on *Supply*



If  $(C, D_3, I_4)$  inserted into *Supply*, FKC not satisfied anymore

DB may enforce satisfaction, e.g. by automatically inserting  $(I_4, \text{NULL})$  into *Articles*

This *NULL* (a null value) represents **an uncertain data value**

**A full, precise logic of the combination of certain and uncertain values has not been implemented in commercial DBMSs, yet** (this applies to the SQL Standard too)

- A view is a relation defined in terms of the base, material relations
- We introduce a new relation name (i.e. a new predicate), and its extension is defined by a query
- A query with a name ...
- The extension can be computed from the definition, but it does not make it into a permanent table

The extension is commonly virtual, and computed upon request and for a session

For the database on page 18, we may introduce a new predicate, whose extension is defined by:

$Compltem(x, z): \exists y \text{ Supply}(x, y, z)$

(More precisely, the definition is:  $\forall x \forall z (Compltem(x, z) \leftrightarrow \exists y \text{ Supply}(x, y, z))$ )

- This view is a particular perspective (view) of table *Supply*
- We do not care about the recipients, as long as they exist
- That is our view of the database (of the relation)
- A view of the database from the perspective of a particular user or group thereof
- We can use this relation name in queries  
E.g. for those providers of item  $l_4$ :  
$$Q''(x) : \text{Completem}(x, l_4)$$
- Data in a DB can be seen in different ways by different users, by different specialized (sub)databases
- For example, starting from the DB on slide 18, a particular user may only see “receivers together with the classes of articles they receive”

Shipment	Receiver	Class
	D1	K
	D2	K
	D2	H

- This particular user does not see the entire database, because it is not useful, relevant, allowed, ...
- Or the user considers the new relationship as particularly relevant
- Usually **virtual relation**
- It will last for a session with the DBMS where it was defined  
Unless it is stored as a physical relation, i.e. **materialized**
- During the session, its contents will be kept in a temporary table
- The **view has to be maintained**  
i.e. kept up-to-date wrt. relevant changes on underlying base relations

- How to specify the view?
- There is not much difference between a view and a query
- We can define it by means of a query in predicate logic (relational calculus)

$Shipment(x, y) : \exists u \exists v (Supply(u, x, v) \wedge Articles(v, y))$

(free variables  $x, y$  receive the answers as values)

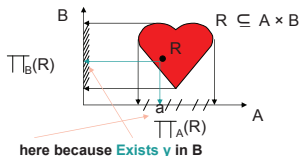
- SQL allows to pose such a query and introduce a name for the view (i.e. the answer set) into the DB

```
CREATE VIEW Shipment AS
SELECT      Receiver, Class
FROM        Supply, Articles
WHERE       Supply.Item = Articles.Item
```

**A query with a name!** Containing a join and a projection

Notice that existential quantifiers capture relational algebra projections

- A relation  $R$  with attributes  $A$  and  $B$  with the following extension in instance  $D$ :



R	A	B
	a	b
	c	b
	a	d

- A view that is the projection of  $R$  on  $A$ :  $P_A(x): \exists y R(x, y)$
- A view that is the projection of  $R$  on  $B$ :  $P_B(y): \exists x R(x, y)$
- Their extensions on  $D$ :  $P_A(D) = \{(a), (c)\}$   
 $P_B(D) = \{(b), (d)\}$
- Their definitions in SQL, resp.:

```
CREATE VIEW P-A AS
SELECT A
FROM R
```

```
CREATE VIEW P-B AS
SELECT B
FROM R
```

- Projection is on attribute in SELECT; the others, the omitted ones, are filtered out

- **View Maintenance:** How to update the view when the underlying database is updated

To keep material DB and view synchronized? (according to its definition)

- Complete re-computation: throw away the old “contents” of the view and compute it again from scratch using its definition (as a query) Expensive if view is large ...
- Incremental update: compute only the **relevant changes**, the “deltas” (there are some techniques)

For example, if we insert  $(I_4, S)$  into *Articles*, it is good enough to “add”  $(D_3, S)$  to the extension of *Shipment*

- Before considering updating the view, detect if the updates on the base relations are **relevant** to the view

- Wrt. relevance of an update on base tables for a view:

- If yes, then an incremental update of the view can be attempted
- Otherwise, the update of the base relation is performed, but ignored for the view
- For example, for the view on page 34, the insertion of  $(C, D_4, I_4)$  in relation *Supply* in page 18 is (potentially) relevant, but with no effect

The insertion of  $(D, D_3, I_1)$  is relevant and does have an impact

- Relevance is determined mainly by the kind of view definition and the kind of update

Less by the actual update and the actual state of the database

- E.g. updates on values for attribute *Receiver* are irrelevant for the *Completem* view

By the kind of view definition (uses a projection), and the kind of update (change of value in a filtered attribute)



- Relevance of updates on base tables can be determined through a syntactic analysis of the view definition
- The same applies to relevance of updates for IC maintenance

The syntactic form of the IC can be used

For example, the syntactic form of an FD (cf. page 31) tells us the deletions are irrelevant (why?)

- Database maintenance and view maintenance are closely related problems; and share techniques
- Another related interesting problem: Update the DB through the view

Update the view and ramify changes to underlying DB

- Not always possible or a deterministic process
- Database updates through views is a complex, important, and not completely solved problem
- Even more complex if there are ICs on the base relations
- And far from being implemented in commercial DBMSs

- For an example, if we insert  $(C, I_6)$  into the view *Compltem*, the change has to be propagated to the base tables

We may insert the tuple  $(C, \text{NULL}, I_6)$  into *Supply*

- How to propagate to the DB the insertion into *Shipment* of the tuple  $(E, L)$ ?
- Missing information again? Maybe NULL values?

A join via NULL values?

- What if some of the attributes with missing information is (part of) a key? (nulls in them not allowed by RDBMSs)

(Key: minimal set  $K$  of attributes of a relation  $R$  such that all the other attributes, say  $Y$ , of  $R$  functionally depend upon, i.e.  $R : K \rightarrow Y$ )

We cannot insert null values everywhere ...

# Active Rules

- Commercial DBMSs offer little support for database maintenance, i.e. for keeping ICs satisfied
- Only a limited class of ICs can be defined with the schema, and automatically maintained satisfied by the system  
E.g. key constraints, not-NULL constraints, referential ICs, ...  
But not arbitrary FDs, etc.
- Also very **limited in terms of how to maintain** those that can be declared/maintained
- **What to do then?**
  - Keep our pet ICs satisfied via application programs that interact with the DB
  - **Store in the DB (stored) procedures that do the job**
- Stored procedures can be quite general, not only for IC maintenance
- They can be explicitly invoked or executed automatically when something happens in the DB
- Active rules, aka. triggers, are of the latter kind

- In abstract terms, active rules have three consecutive components: *Event-Condition-Action* (ECA) rules
  - When an *Event* happens (e.g. an update of a certain kind on the DB), and
  - A *Condition* is true at the current DB state (e.g. an IC is violated, which can be detected through a query), then
  - An *Action* (e.g. a DB update or a message to the external world) is automatically executed
- For example, to keep the referential IC in page 19 satisfied:  
$$\forall x \forall y \forall z (Supply(x, y, z) \rightarrow \exists w Articles(z, w))$$
  
(assuming that IC was satisfied before the update)
  - (Relevant) **Event**: insertion of  $\langle a, b, c \rangle$  into *Supply*
  - **Condition**:  $V(x, y, z): Supply(x, y, z) \wedge \neg \exists w Articles(z, w)$   
true for  $\langle a, b, c \rangle$ ?  
(violation view for the IC becomes non-empty, but it should be)
  - **Action**: (If yes,) insert  $\langle c, NULL \rangle$  into *Articles*
- **Exercise**: Not the only way to violate the IC. Design an ECA rule for the other case.

# Active Rules

- In this example, relevant events are:
  - An insertion into *Supply*, and
  - A deletion from *Articles*
- Being those relevant updates part of the *Event*, the *Condition* asks if a violation is produced (it catches violations); and, if *yes*, the *Action* could be “reject the update” or a compensating update (to satisfy the IC)
- The relevant updates (events), conditions, and actions for a given IC can be computationally derived from the syntactic form of the IC
- Triggers can be shared; and they are useful for, among other things, IC maintenance, view maintenance, etc.
- Active rules can be used also for business applications
- Capturing business rules for the application domain
- **Example:** If the stock (or inventory as shown in a table) goes below a certain pre-specified threshold, insert a request for resupply into the *Orders* table