Chapter 0: Review of Relational Data Management and Databases

(and a bit more)

Leopoldo Bertossi

Data Models, Models in General

- 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

Examples of Models (among many ...)

Example from (very basic) Physics

The model is formed by the motion equation, for the position as a function of time:

$$p(t) = 0.5 \times t \ + \ p(0)$$

plus the initial observation: $p(0) = 2$
This is the model
We do not explicitly include all possible pairs (position, time)
That is implicit
We can obtain from the model: $p(6) = 5$

Example from Logic

Consider the Aristotelian argument:

> *"If Socrates is a man, then Socrates is mortal. Socrates is a man. Then, Socrates is mortal."*

It is expected to be a *valid* argument, i.e. always true

Actually more due to its logical structure than to the particular properties, e.g. being mortal, and individuals involved, e.g. Socrates

All this could be expressed in propositional logic, as follows:

1. Denote the most basic and atomic propositions involved, i.e. not decomposable in sub-propositions, by means of propositional variables

   P: "Socrates is a man" Q: "Socrates is mortal"

2. Use symbolic logical connectives to combine the propositional variables, to symbolically express the first statement above (among many other possible statements): $(P \rightarrow Q)$

The model in this case becomes the following knowledge base expressed in propositional logic:

$$((P \to Q) \land P)$$

Or, equivalently, the set of formulas $\{(P \to Q),\ P\}$
The implicit knowledge should be Q, in the sense that

$$((P \to Q)\ \land P)\ \to\ Q)$$

is an always true formula, i.e. a tautology,
In fact, this formula is always true, for any truth values (0 or 1) assigned to P and Q
This can be easily checked with a truth table (Do it!)

Or by expressing it in an logically equivalent formula:

$$\begin{aligned} &\equiv \ \neg((P \to Q) \ \wedge P) \vee Q \equiv (\neg(P \to Q)) \ \vee \neg P) \vee Q \\ &\equiv (\neg(\neg P \vee Q)) \ \vee \neg P) \vee Q \equiv \neg(\neg P \vee Q) \vee (\neg P \vee Q) \end{aligned}$$

The last formula is of the form $\neg p \vee p$, which is clearly always true, the most common form of tautology!

Now consider the argument:

> *"All men are mortal. Socrates is a man. Then, Socrates is mortal."*

It should be again a valid argument, but it cannot be expressed as such in propositional logic

Let's try as above

1. Propositional variables: P: "All men are mortal"
   Q: "Socrates is a man" R: "Socrates is mortal"
2. The argument in propositional logic?
   $(P \wedge Q) \rightarrow R$

No longer a tautology: It is logically equivalent to
$\neg(\neg P \vee Q) \vee R \equiv (P \wedge \neg Q) \vee R$

If R is false and Q true, the formula becomes false

We need a more expressive logic: predicate logic
The argument above involves general, non-instantiated concepts (properties, predicates, attributes), e.g. "being mortal"

1. Now start by introducing symbolic predicates: $Ma(\cdot)$, $Mo(\cdot)$
   A constant to denote (name) Socrates: s
   Variables: $x, y, \ldots$ (implicitly ranging over the underlying domain of discourse)
   A universal quantifier: $\forall$
2. Now the argument as a formula of predicate logic:
   $(\forall x(Ma(x) \rightarrow Mo(x)) \wedge Ma(s)) \;\rightarrow\; Mo(s))$

This logic is more expressive than propositional logic, and the one we use in databases and many areas of data management!

Data Models, ER, Relational Model

- 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.

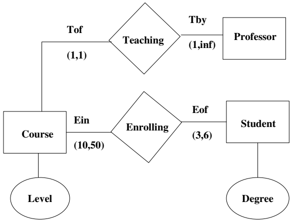
Data Models, ER, Relational Model

- 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?

Data Models, ER, Relational Model

- An Entity/Relationship (ER) Model:

  A graphical "language" that uses diagrams to model data

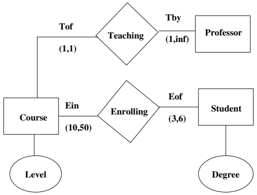
  - Entities: represented by boxes; they correspond to classes of data items of a same kind
  - Relationships: by diamonds; they relate (data items from) different entities
  - Attributes: by ovals hanging from entities; representing properties (of elements) of an entity

    May also hang from relationships
  - Cardinality constraints: label links between relationships and entities

- Do not take edge (link) names too seriously for now
- They become relevant when ER is translated to an ontology
- Intuitively, for entity *Course*, and through the relationship *Teaching*, the entity *Professor* becomes an "attribute" of *Course*

  Hence the "TBy" (taught by). Etc.

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*

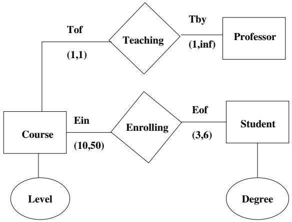
  We could also have an attribute *Term* hanging from the *Enrolling* relationship

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

Data Models, ER, Relational Model

- *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
  And the model is modeling the external reality
  The more elements (that semantically correct wrt. the external reality) we add to the model, the closer we are to the external reality

Data Models, ER, Relational Model

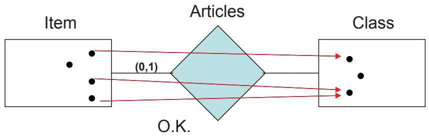


- Cardinality constraints are used to capture more meaning

  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

Data Models, ER, Relational Model

- Cardinality constraints can be understood as restrictions on the mappings between entities through the relationship



  - 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 ...

Data Models, ER, Relational Model

- Relational Model of Data: (back to example on page 11)
  - 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:

Data Models, ER, Relational Model

(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 18)

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

Relational Databases

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 18

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

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

Relational Databases

- 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

Relational Databases

- 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 11 shows a database instance for the schema defined on page 19

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
  - Nor about what internal data structures and access methods are used

Relational Databases

- 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.

Relational Databases

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

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

  Algebraic, imperative

- Relational calculus: $\exists y \exists z\ Sales(x, y, \text{‘cd’}, z)$

  Logical, declarative

  Variable x is not quantified (it’s free), and its possible values are the query 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 ...)

Relational Databases

Example:

Supply	Company	Receiver	Item
	C	D_1	I_1
	D	D_2	I_2

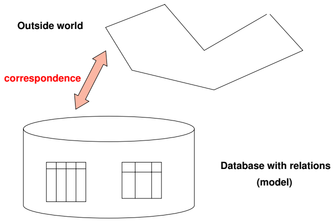
Articles	Item	Class
	I_1	K
	I_2	K

- 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

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 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)

Relational Constraints

- 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 ...

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

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 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)

Relational Models

- 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

Relational Models

- 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)

Relational Models

- 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

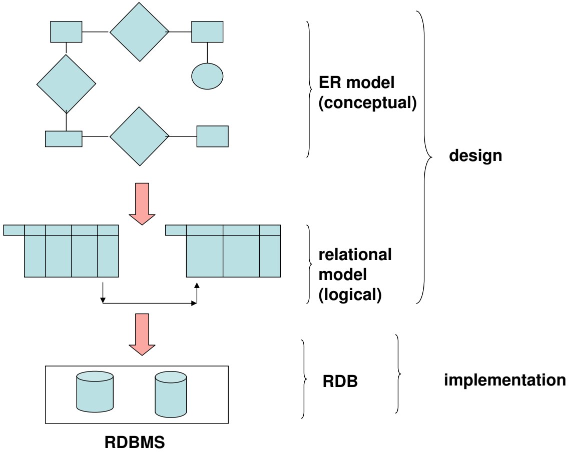
  By creating the schema
- Finally, the database is populated (with data)

  Obtaining an instance
- Instances change frequently

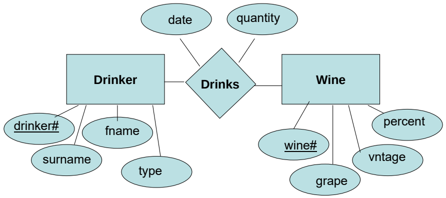
  Schemas not so much

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

Relational Models



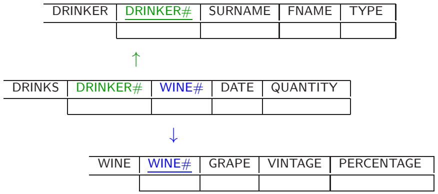
Example: From an ER model to a relational schema



- In this ER model the attribute `dinker#` appears underlined
- This expresses a new kind of semantic constraint: this attribute becomes a unique identifier
- In the sense that any two instances of entity `Drinker` share the same value for attribute `dinker#`, then they have to share the same values for all the other attributes of `Drinker`

Similarly for `Wine#` in entity `Wine`

The ER model above is transformed into the relational schema:


DRINKER | DRINKER# | SURNAME | FNAME | TYPE
↑
DRINKS | DRINKER# | WINE# | DATE | QUANTITY
↓
WINE | WINE# | GRAPE | VINTAGE | PERCENTAGE

- The arrows indicate referential integrity constraints introduced into the schema
- E.g. any drinker number in relation `Drinks` must appear also in relation `Drinker`
- Attribute `Drinker#` appears underlined in relation `Drinker`, indicating a relational semantic constraint (inherited from the one in ER)
- In this case, a key constraint: that attribute functionally determined the others in the relation (more on this coming)

Queries

For the the instance

Deposit

branch	acc#	clientn	balance
Carleton	101	Jim	500
Downtown	215	Sandy	700
Barrhaven	304	Alvin	1300

Client

clientn	cladd	neighcl
Jim	101 Queensbury	Barrhaven
Sandy	40 Stone	Nepean
Hernandez	15 Laurier	Downtown
Alvin	17 Clyde	Altavista
John	89 Case	Centrepoint

"give me the addresses with balances of the clients who have a balance higher than 600"

Answer:

40 Stone	700
17 Clyde	1300

The answer is a set of tuples, a new relation (extension)

We can say that a query is a mapping that sends DB instances to new DB instances (possible with a different schema)

Several issues:

- How to specify a query?
  How to write it?
  In what language?
- How expressive is the chosen query language? Can it capture/represent any reasonable query?
- What is the precise meaning of a query?
- What is the precise meaning of a query answer?
- How to compute the answer?

There are several query languages for RDBs
Some more used in practice than others
But those of a more theoretic nature are the basis for the languages most used in practice

The distinction between declarative vs. procedural query languages is always relevant

The former express what the user wants to obtain from the database, the latter express a particular way to compute the answer

Queries: Relational Algebra

Idea: Relations are sets (subsets of cartesian products) constructed on top of other sets (domain or subdomains)

Query answers are new relations

Thus, in order to obtain new relations (e.g. query answers) do set-theoretic algebra on existing relations

Operate on sets and relations in order to obtain new sets or relations

The Relational Algebra (RA)

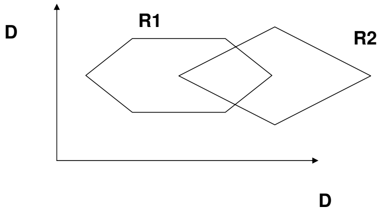
- Provides algebraic operations over relations that produce new relations
- Operations based on set-theoretic operations
- Some of those operations come directly from set theory
  Others are specific, *ad hoc*, for the RA
  The latter are applicable to relations (as opposed to sets in general)
- Provides a procedural query language for RDBs (because it is based on explicit operations)
- The RA is one of the strengths of the relational model
- RA can be used to give a precise, set-theoretic semantics to other query languages

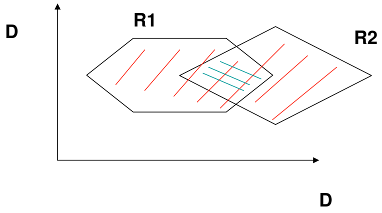
Queries in RA:

- It is possible to answer the query by applying a sequence of algebraic (relational) operations starting from the original database instance
- The query in RA becomes a finite sequence of algebraic operations to be executed (in sequence) on the given instance, i.e. an algebraic formula
- Even if the RDBMS offers a different query language, e.g. a declarative one, a query will be compiled into a sequence of algebraic operations on the DB

Summary of basic operations of RA:

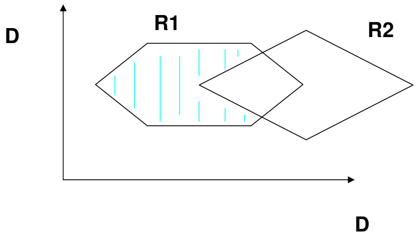
- *Union* and *Intersection*: $R_1 \cup R_2, \quad R_1 \cap R_2$
  Can be applied to similar relations (i.e. with the same (sub)schema; in particular, same arity and data types) as usual sets
- *Difference*: $R_1 \setminus R_2$
  Again, for similar relations, as normal sets
- *Product*: $R_1 \times R_2$
  This is essentially the cartesian product of two relations taken as normal sets; not necessarily with the same schema
  E.g. for $R = \{(a, b), (c, d)\}, \quad S = \{(1, 2), (2, 3)\}$
  $R \times S = \{(a, b, 1, 2), (a, b, 2, 3), (c, d, 1, 2), (c, d, 2, 3)\}$





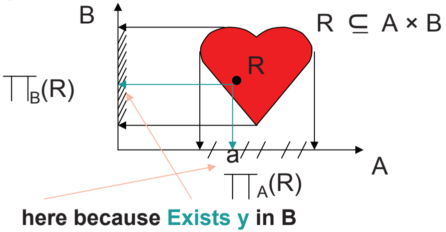
$R1 \cup R2$

$R1 \cap R2$



R1 \ R2

- *Projection*: $\Pi_A R(\cdots, A, \cdots)$, i.e. the projection of relation R on attribute A



Here, A is one of the attributes of R
The projection could be on several attributes of R
This is a unary operation: takes one relation as input (the previous ones are binary)

This is an operation special for relations

It deletes, ignores, filters out entire “columns” from a relation

Projects R over one (or several) “coordinates” (attributes)

It generates a new relation, with a subset of the attributes (columns)

Its logical counterpart is the existential quantification

For the relation in the figure:

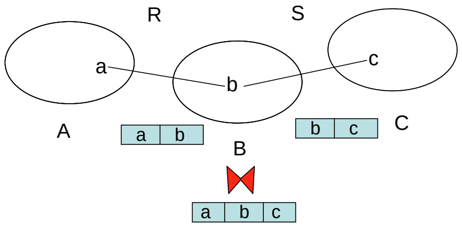
$$\Pi_A R(A,B) \;=\; \{a \in A \mid \text{ it exists } b \in B \text{ such that } (a,b) \in R\}$$

Similarly, we can use $\Pi_{AB} R(A,B,C)$, dropping attribute (column) C

- *Selection*: $\sigma_{<condition>}(R)$
  Unary operation, special for relations
  Selects the tuples of the relation R that satisfy the *condition*
  The condition can be expressed in a (limited) logical language
  It generates a new relation, with the same attributes, but possibly fewer tuples (rows)
  Example: $\sigma_{Balance>3K}(Deposit)$ keeps only those tuples of *Deposit* that show a balance greater than 3K

- *Join*: $R_1 \bowtie R_2$
  A binary operator, essential in RA
  In its simplest form it allows to compose two relations through the values in common taken by a distinguished attribute that is shared by the two relations
  (Or two different attributes, one in each relation, but with same data type or domain)
  Similar in spirit to the operation of composition of two relations as in set theory:
  $R \circ S := \{(a, b)|$ there is c with $(a, c) \in R$ and $(c, b) \in S\}$
  It is essential for combining tables in natural way, without appealing to the possibly large and computationally expensive *product* of them

This is the basic, natural join; for which there are generalizations



Example: Again: *"give me the addresses with balances of the clients who have a balance higher than 600"*

Deposit

branch	acc#	clientn	balance
Carleton	101	Jim	500
Downtown	215	Sandy	700
Barrhaven	304	Alvin	1300

Client

clientn	cladd	neighcl
Jim	101 Queensbury	Barrhaven
Sandy	40 Stone	Nepean
Hernandez	15 Laurier	Downtown
Alvin	17 Clyde	Altavista
John	89 Case	Centrepoint

It can be expressed in RA through the formula:

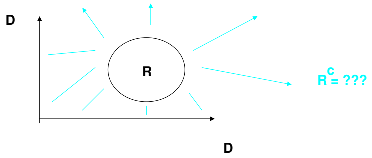
$$\Pi_{\text{cladd, balance}}(\sigma_{\text{balance} > 600}(\mathit{Deposit}) \bowtie_{\text{clientn}} \mathit{Client})$$

$$= \begin{array}{|c|c|} \hline \text{40 Stone} & 700 \\ \text{17 Clyde} & 1300 \\ \hline \end{array}$$

The RA formula

$\Pi_{\text{cladd, balance}}(\sigma_{\text{balance} > 600}(\mathit{Deposit} \bowtie_{\text{clientn}} \mathit{Client}))$ also works, but its evaluation is less efficient

Notice: There is no (set-theoretic) *Complement* operation in RA (as found in set theory)



In principle, there could be, given that relations are sets, but
What is the "meaning" of the complement of a relation?
Actually, it could be infinite, because the DB domain is possibly infinite
The difference ($\smallsetminus$) is only a *relative complement*, relative a given relation

`Deposit`

branch	acc#	clientn	balance
Carleton	101	Jim	500
Downtown	215	Sandy	700
Barrhaven	304	Alvin	1300

Which could be the *tuples that are not in Deposit*?
Which of those make sense?
This query is not admissible in relational databases

Example: Relational schema:
$Frequents(Drinker, Bar)$, $Serves(Bar, Beer)$, $Likes(Drinker, Beer)$
Express in RA the query about the drinkers who do not frequent any bar that serves some beer they like

1. $R_1 = Serves \bowtie_{Beer} Likes$
2. $R_2 = R_1 \bowtie_{Drinker,Bar} Frequents$
3. $R_3 = \Pi_{Drinker}(R_2)$

   Up to here those drinkers who frequent bars that serve beer they like
4. $R_4 = \Pi_{Drinker}(Likes)$
5. $R_5 = \Pi_{Drinker}(Frequents)$ (having just this one was also O.K.)
6. $R_6 = R_4 \cup R_5$

   With these last three steps, we collect all drinkers
7. $R_7 = R_6 \smallsetminus R_3$ (the only form of "negation" we have in RA)

Exercise: Put some data in the schema and check it works!
Notice: the query is general, independent from any specific instance, and depending only on the schema

Queries: Relational Calculus

- 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 28, we want to obtain the classes of articles that are provided by company D

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

$\overset{\text{query}}{\longmapsto}$

Answer	*Class*
	K

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

Queries: Relational Calculus

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

$\overset{\text{query}}{\longmapsto}$

Answer	*Class*
	K

- 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:

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

Queries: Relational Calculus

- $\exists y \exists z (Supply(D, y, z) \wedge Articles(z, x))$ (*)
  - 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

Queries: Relational Calculus

- 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
```

Queries: Relational Calculus

- Another SQL query, for the schema $Accounts(Account\#, Name, Balance)$:

```
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):

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

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

Example: (continued from page 54)

The query expressed in SQL

```
SELECT F.Drinker
FROM   Frequents F
WHERE  F.Bar NOT IN (SELECT S.Bar
                     FROM   Serves S, Likes
                     WHERE  Likes.Drinker = F.Drinker AND S.Bar = Likes.Bar)
```

- Also general: we do not "see" the (whole) contents of the DB before posing the query
- Queries provide the way to obtain values from the DB

How can we express the same query in Relational Calculus?

Queries: Evaluation

- A RDBMS is able to take a query (say 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 (compare page 51)

    E.g. if possible, better apply a selection before a join, because the join -an expensive operation- is reduced
  - Usually done automatically and internally by the DBMS

- Semantic query optimization:

  Take advantage of explicit ICs to optimize query answering

  Hence “semantic QO”: ICs convey semantics

  - 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
  - Some systems provide basic SQO

    Others also on the basis of ICs provided by the user as “informational ICs” (e.g. IBM DB2) that are not necessarily checked by the DBMS (warning! more on this coming)
  - How can SQO automated? $\rightarrow$ Use logic to express queries and ICs, and combine them symbolically

Integrity Constraints (revisited)

- 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 that 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

Supply	Company	Receiver	Item
	C	D_1	I_1
	D	D_2	I_2

Articles	<u>Item</u>	Class
	I_1	K
	I_2	K

- 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)"

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

- Notice that the schema determines the logical language
- This same language can be used to express queries and ICs
- Since both are symbolic, they can also be symbolically (i.e. syntactically) and automatically combined, e.g. for semantic query optimization
- It could also be expressed in RA as a containment of projections (but much less common) Do it!

- Example continued
  - We can also impose the condition that *Item* is a key for relation *Articles*
    I.e. all attributes of *Articles* functionally depend upon *Item*
    ($Articles\colon\ Item \rightarrow Class$)
  - Symbolically as on page 63
  - Exercise: Give another example of a key and its logical formulation
  - The combination of the two is a foreign key constraint on *Supply*: Its attribute *Item* is the key in a foreign relation

Interlude on keys vs. functional dependencies (FDs):

ArticlesNew	<u>*Item*</u>	*Class*	*Price*
	I_1	K	10
	I_2	H	20

Key Constraint is satisfied

Here, two FDs: $Item \to Class$ and $Item \to Price$

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

ArticlesNew	*Item*	*Class*	*Price*
	I_1	K	10
	I_2	H	20
	I_2	H	30

O.K. if only FD: $Item \to Class$

Item is not a key for the 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*

Supply	Company	Receiver	Item
	C	D_1	I_1
	D	D_2	I_2

Articles	Item	Class
	I_1	K
	I_2	K

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

DB may enforce satisfaction, e.g. by automatically inserting $(I_4, NULL)$ 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)

Main ways to maintain (the consistency of) the DB:

- Some limited mechanisms provided by the DBMS, e.g. rejection of updates
- Some limited mechanisms the user can specify together with the declaration of the IC, e.g. cascaded deletion for ref. IC, or insertion of null (as on this page)
- User-defined triggers that are stored in the DB; they react spontaneously (coming)
- Through application programs

  DBs are seldom used in isolation

  Users run programs that interact with the DB, querying it, massaging and inspecting data outside the DB, i.e. at the program level, putting new data inside the DB, etc.

  Verification and maintenance of ICs can be done from the application program

Views

- 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 28, we may introduce a new predicate, whose extension is defined by:

$$\mathit{CompItem}(x, z): \quad \exists y\ \mathit{Supply}(x, y, z)$$

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

Views

- 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 I_4:

  $$Q''(x) : \quad CompItem(x, I_4)$$
- 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 28, a particular user may only see "receivers together with the classes of articles they receive" (see def. on page 73)

Views

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

- Many other uses of views:
  - Privacy, security (give access to views of DB, not the whole DB)
  - Query optimization: Reuse cached contents of view to answer new queries (whenever possible)

    "Query answering using views" (coming)
  - Catch potential inconsistencies wrt. ICS (coming)
  - Data integration (coming)

Views

- 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
```

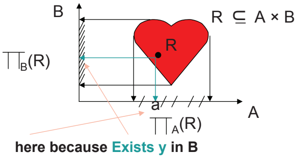
A query with a name! Containing a join and a projection
Notice that existential quantifiers capture relational algebra projections

The view can be used in queries, e.g. about receivers of items in class K:

```
SELECT Receiver
FROM   Shipment
WHERE  Shipment.Class = 'K'
```

Views

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

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

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

Active Rules

- 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 (intended) 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 an internal query)
  - Then, an *Action* is automatically executed, e.g. a compensating DB update or a rejection/warning message to the external world

    A more complex stored procedure could be invoked by the rule action

Active Rules

- For example, to keep the referential IC on page 29 satisfied:

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

  Assumption: the IC is satisfied so far (before the update)
  An inductive process to check IC and keep it satisfied
  Based on analysis of relevant updates (as already seen)

  - (Relevant) Event: insertion of $\langle a, b, c \rangle$ into *Supply*
  - Condition: $V(x,y,z)$: $Supply(x,y,z) \land \neg \exists w Articles(z,w)$
    true for $\langle a, b, c \rangle$?

    Violation view (VV) for the IC becomes non-empty

    IC satisfied iff associated VV is empty
  - Action: (If yes,) insert $\langle c, NULL \rangle$ into *Articles* (as compensating update); uses information from VV (or Condition, in general)

- Exercise: Not the only way to violate the IC. Design an ECA rule for the other cases.

Active Rules

- In this example, relevant events are:
  - Insertions into *Supply*, and
  - Deletions from *Articles*
  - Changes of attribute values (some)
- Being those relevant updates part of the *Event*, the *Condition* asks if a violation is produced

  It catches violations through a violation view

  If *yes*, the *Action* could be "reject the update" or issue 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

Active Rules

- Triggers can be shared by users and applications
- 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/from the application domain

Exercise: 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

Create a small DB with its schema to make this more concrete

Indicate the ECA components