Relational Calculus

- The Relational Calculus is a declarative query language for RDBs

  It comes from the origin on RDBs in Predicate Logic

- Example: (of page 36 cont.) *“Classes of articles 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

- This query (as a mapping) can be expressed as a relational calculus query (formula):

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

- With intended semantics:

$$\underbrace{\exists y \exists z}_{\text{there are } y,\, z} \; (Supply(\underbrace{D}_{\text{company as a constant}} \underbrace{, y, z) \underbrace{\wedge}_{\text{and}} Articles(z,}_{\text{an implicit join}} \underbrace{x}_{\text{free variable to retrieve answers}}))$$

- K is an answer, because the logical formula (*) is true in the given database instance when x takes the value K
- More specifically, 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 database instance
- A completely declarative query!
- Also symbolic, follows a precise syntax (grammar)

  It is also machine processable!
- Same query now in RA: $\Pi_{Class}\sigma_{Company='D'}(Supply \bowtie_{Item} Article)$
- RA and RC are provably equally expressive

  A RDBMS evaluates query (*) by translating it into an "optimized" RA query

SQL

- RA and RC are the basis for the common and standardized query language for RDBs: SQL (Structured Query Language)

- The previous query as an SQL query:

$$\text{SELECT Class} \quad \text{FROM Supply, Articles} \quad \text{WHERE } \underbrace{\text{Supply.Company} = \text{'D'}}_{\text{implicit relational selection}} \text{ AND } \underbrace{\text{Supply.Item} = \text{Article.Item}}_{\text{implicit relational join}}$$

"Selecting (values for) *Classes* from table *Articles* in combination with table *Supply*, when the *Company* attribute from table *Supply* takes value *D* and the *Item* attribute takes the same values in both tables"

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

(attribute selection)	SELECT Name
(table selection)	FROM Accounts
(condition)	WHERE Balance > 10,000

Asking for values for *Name* attribute of *Accounts* table of those customers who have a balance greater than 10,000

- Last query in relational calculus:

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

- SQL queries can be fully declarative (see previous examples)
- Also possible to bring RA operations into SQL queries
- SQL can also be used to:
  - Create, modify and query metadata: the schema
  - Update the relations
  - Express/impose Integrity Constraints
  - Define views
  - ...
- More on all this coming ...

Integrity Constraints (revisited)

- Relational Calculus can also be used to state ICs
- E.g. functional dependencies (FDs):

"items cannot be associated with more that one class"

$$\underbrace{\forall x \forall y \forall z}_{\text{for all possible values}} \; (\; \underbrace{Articles(x, y) \wedge Articles(x, z)}_{x \text{ is associated to values } y \text{ and } z} \; \rightarrow \; \underbrace{y = z}_{y,\, z \text{ must be equal}} \;)$$

(saying that for every article, x, if it is associated to any two articles, y, z, then they are the same)

- A sentence: a logical formula without free variables

  Can be seen as a binary query: It is true (1) or false (0) in the instance

- A declarative IC! It states how things should be

  A separate computational mechanism has to keep it satisfied, i.e. true in the database instance, even under updates

- A symbolic constraint, which is quite useful: Allow for symbolic computations with them (examples later)

- Example:

Supply	Company	Receiver	Item
	C	D_1	I_1
	D	D_2	I_2

Articles	Item	Class
	I_1	K
	I_2	K

$\downarrow$ ______________ $\uparrow$

- RC can express this *referential IC* from *Supply.Item* to *Articles.Item*: *"every item value in the former must appear in the latter (the official list of items)"*

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

(for every item, z (and accompanying values, x, y), if it appears in *Supply.Item*, then it appears in *Articles.Item* accompanied by some class value, w)

- We can also impose the condition that *Item* is a key for relation *Articles* (as we did before, even in RC)

$$Articles: \ Item \rightarrow Class$$

- The combination of the two ICs is a foreign key constraint on *Supply*

  Its attribute *Supply.Item* is the key in a foreign relation *Articles*

- Database maintenance is the problem of keeping the database consistent

  That is, satisfying the specified ICs when it undergoes updates

  Many important issues around this problem (we will come back)

- A more general remark: Notice from the examples above that the schema determines the RC language to use

- The same logic-based language can be used to express queries and ICs

- Since both are symbolic, they can be syntactically and computationally combined

  This can be very useful, e.g. for query optimization (examples later)

- RC is the language of choice to express ICs (or its SQL incarnation)
  (The referential IC could be expressed in RA as a containment of projections, but is uncommon Try to do it!)

Views

- A view is a defined relation

  In terms of the base, material relations (the tables)

- A new relation name (i.e. a new predicate) is introduced

  Its extension is defined by (as the result of) a query

  It is a query with a relation name

- The view extension can be computed from the definition

  But it does not become a permanent (materialized, physical) table

- The extension is virtual

  Computed upon request, and for a session

  Not permanently stored in the DB

  Unless it is explicitly materialized (not very common)

Example:

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
	I_4	H

- Introduce a new predicate, whose extension is defined by:

$$\underbrace{CompItem}_{\text{new 2-ary predicate}}(x, z): \quad \exists y \; Supply(x, y, z)$$

- Precise definition is:

$$\forall x \forall z (CompItem(x, z) \quad \underbrace{:\longleftrightarrow}_{\text{LHS defined by RHS}} \quad \underbrace{\exists y Supply(x, y, z)}_{\text{in terms of existing elements}})$$

- This view is a particular perspective (view) of table *Supply*
  We do not care about the receivers as long as they exist
- That is our view of the database (or of the relation)
- A view of the database from the perspective of a particular user or group thereof

- Its virtual extension can be computed (and kept in main memory)

CompItem	Company	Item
	C	I_1
	D	I_2
	C	I_4

- This particular user does not see the entire database: not useful, irrelevant, disallowed, ...

  Or user considers the relation as particularly relevant

- A virtual relation that will last for the session with the DBMS

  During the session, its contents will be kept in a temporary table (unless it is stored as a physical relation, i.e. materialized)

- Many other uses of views:
  - Privacy, security (give access to views, not to whole DB)
  - Query optimization: Reuse cached contents of view to answer new queries (whenever possible)
  - Query answering using views
  - Monitor internal processes, e.g. catch potential inconsistencies w.r.t. ICs (see next section)
  - Data integration

Example: Want this view (result)

Shipment	Receiver	Class
	D_1	K
	D_2	K
	D_2	H

- How to specify the view?
- Not much difference between a view and a query

  Its definition via a query in RC:

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

- SQL allows to define the view including its defining query

```
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 (compare with (*))

  Existential quantifiers capture relational algebra projections

- View can be used in queries: *"receivers of items in class K"*

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

Active Rules, Triggers

- 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 by the system

  E.g. Key Constraints, Referential ICs, Not-NULL Constraints
  (disallowing certain attributes from having missing values)

  Also very limited on how to maintain them

  For others there is no built-in support, e.g. arbitrary FDs

- How to keep then satisfied?
  - Via application programs interacting with DBMS
  - Store in the DB an active procedure that does the job

    Reacting (running) automatically when there is something to do: Active Rules! (a.k.a. Triggers)

- An AR can be seen as a stored procedure

  Can be explicitly invoked or executed automatically by DBMS when something happens in the DB

- They are defined with SQL (syntax and semantics depending on vendor)

- In abstract terms, ARs have three components:

  Event-Condition-Action (ECA rules)

  - When an *Event* happens (or is about to) in the DB

    E.g. an intended update of a certain kind on a table

  - And a *Condition* is (about to be) true in the DB

    E.g. a violation of the IC (which can be detected through an internal, pre-specified query)

  - Then, an *Action* is automatically executed

    E.g. a compensating DB update or a rejection/warning message to the external world

    (Or calling a more complex stored procedure could be invoked)

Example: Keep the referential IC satisfied under insertions

(c.f. page 64)

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

- Assumption: IC is satisfied before the insertion
- Only relevant "insert" *Event:* Insertion into *Supply*, e.g. of tuple $\langle a, b, c \rangle$
- *Condition:* The insertion creates an inconsistency

  Has to be checked via a pre-specified query (basically the same query all the time)
- Define a violation view that catches those inconsistencies:

$$V(x,y,z): \ Supply(x,y,z) \ \wedge \ \underbrace{\neg \exists w Articles(z,w)}$$

it is not the case that z appears in *Articles* accompanied by some item value w

Is this true for $\langle a, b, c \rangle$?

- If it is, i.e. $\langle a, b, c \rangle$ is answer to the view query, equivalently belongs to the view

  A flag: A non-empty view! (and it should be)

- Execute the *Action:* Insert $\langle c, \mathtt{NULL} \rangle$ into *Articles*
- A compensating update

  It uses information from the view (value c)

- The one above is not the only way to violate the IC, nor the only way to restore it

  Exercise: Consider the other cases and associated ECA rules

- Triggers can be shared by users and applications
- They are useful in many ways, not only IC maintenance

  There are other "internal" applications

- Also "external" applications, e.g. in Business
- Capturing business rules for/from the application domain (whose data is in the DB)

Exercise: (inventory management) 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