Database design

- Usually designing a database consists of three tasks:
  - conceptual design - what data to include and how these data are inter-related
  - logical design - how the data are presented as logical data structures
  - physical design - how the data are organized as files and indexes.

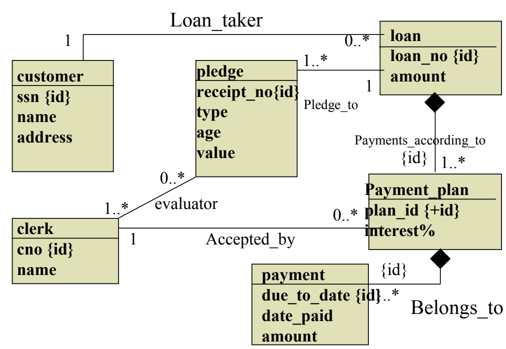
Database design

- Conceptual design produces an abstract model of data to be included in the database
- This model
  - is independent of any database management system
  - reflects the structure of the universe of discourse (I.e. the topic about which data will be gathered)
  - is based on some dedicated modeling technique (Entity-Relationship, UML) - these are discussed in course Introduction to Application Design and Analysis)

Database design

- Conceptual design is actually analyzing the universe of discourse in order to find out which phenomena are such that should be represented as data in the database
- The result of this analysis
  - identifies the types of objects about which data will be collected
  - identifies the properties of objects that will be presented as data items
  - identifies such dependencies among objects and data items that should be reflected in the database

Database design



Database design

- Logical design defines the database for some type of database management system, for example, for relational dbms.
- It considers how the data are represented using the structures offered by dbms:
  - what datatypes to use
  - how to organize the data into tables
  - what are the keys
  - how to connect rows

Database design

- If we had done the conceptual design using E-R or UML technique, there is a straight forward way to transform the obtained data model into a schema of a relational database.

Database design

- Physical design is concerned on how the database is organized as files and what kind of structures to use for efficiency of database processing

Database design - Logical Design

- In the design of relational databases the main issue is to organize the data in relations in a way that avoids redundancy i.e. to store each piece of information only once
- This makes the database easier to maintain
- Storing the same information repeatedly causes many problems
  - storage space is wasted
  - updating data becomes complex
  - modification operations may have unexpected side-effects

Database design - Logical Design

- An example of a table that has redundancy:

EMP_DEPT:

E_no	E_name	E_bdate	D_no	D_name	D_location
1	M.Smith	1.3.59	3	Sales	Helsinki
2	D.Lowe	4.10.40	3	Sales	Helsinki
3	K.Knuth	30.1.66	4	Admin	Lahti
4	B.West	2.5.65	4	Admin	Lahti
5	O.East	10.2.55	6	Production	Helsinki

Key: E_no

if O.East is deleted, also information about Production dept is lost

Location must be repeated

If Admin dept. Moves to Espoo we must update many roles

Database design - Logical Design

- We get rid of the problems with tables

Employee

eNo	eName	bDate	Dept
10	M.Smith	1.3.59	3
20	D.Lowe	4.5.40	3
30	S.Knuth	8.6.66	4
40	B.West	2.4.65	4
50	O.East	1.2.55	6

Department

dNo	dName	dLocation
3	Sales	Helsinki
4	Admin	Espoo
6	Production	Espoo

Database design - Logical Design

- The re-organization was made based on dependencies among data items. We may use the dependencies to determine which data items belong together (into the same table).
- Actually we used only one type of dependency - the functional dependency

Database design - Logical Design

- Functional dependency
- Let columns A and B belong to the same relation schema R. Column A determines column B functionally, if no value of A is associated to more than one value of B (in whatever instance of schema R)
- (to be associated= values are in the same row)
- Notation A ->B
- A may be a single column or a collection of columns

Database design - Logical Design

A	B	C	D
aaa	bbb	ccc	dda
aaa	bbb	cca	ddb
aab	bbc	ccd	ddd
aab	bbc	cca	dde
aab	bbc	ccc	ddc

According to this table instance it seems that

A->B, D->A, D->B, D->C , AC->D, BC->A

Database design - Logical Design

- Dependencies D->A, D->B, D->C are 'true' because each D value is unique,
- Similarly each BC and AC combined value is unique
- A-values are not unique but no A-value appears together with more than one B-value
- We may not, anyhow, determine functional dependencies relying on one instance of a table - the condition must be true in all potential instances

Database design - Logical Design

- Thus the dependencies truly exist, if D-values,and AC-and BC -combined values are unique in all possible instances.
- Let's use a more concrete example with the same template

Database design - Logical Design

Job	Salary	Address	EmpNo
clerk	2000	ccc	10
clerk	2000	cca	20
analyst	3000	ccd	30
analyst	3000	cca	40
analyst	3000	ccc	50

OK: EmpNo->Job, EmpNo->Salary, EmpNo->Address
OK: Job->Salary
Not always: Salary,Address->Job
Not always: Job,Address-> EmpNo

Database design - Logical Design

- EmpNo->Job
  - (each employee has one job)
- EmpNo->Salary
  - (there is only one salary for each employee)
- EmpNo->Address
  - (each employee has only one address)
- Job->Salary
  - (salaries are job specific)
- NO: Salary,Address->Job
  - (if we know employee's salary and address we are able to determine his job)
- NO: Job,Address-> EmpNo

Database design - Logical Design

- There are also other functional dependencies like
  - EmpNo, Salary -> Address
- This is however derivable because
  - EmpNo->Salary and there is a rule saying that
  - if X->Y then XZ->Y for any Z
- There are also other rules (Armstrong axioms) on how to derive dependencies, an important rule is transitivity:
  - if X->Y and Y->Z then X->Z

Database design - Logical Design

- Key and fuctional dependencies
  - The key of a relation may be defined based on functional dependencies as follows
  - **Attribute collection K is the key of relation R if K->X for each attribute X in R and no subset of K has this same property.**
  - Thus the key for relation
  - Emp(Job,Salary,Address,EmpNo) is EmpNo

Database design - Logical Design

- Boyce-Codd normal form (BCNF) is one criteria for a good relational schema (table structure).
- A relation is in Boyce-Codd normal form, if there are no fuctional dependencies X->Y related to it such that X does not contain the key of the relation
- Emp(Job,Salary,Address,EmpNo) is not in BCNF because its key is EmpNo and there is the dependency Job->Salary, where EmpNo is not part of Job.

Database design - Logical Design

- Example
- Shopping(productId, productName, listPrice, buyerName, reduction%, paidPrice, whenMade)

productId -> productName (OK)
productId -> listPrice(OK)
productId -> buyerName (NO)
productId -> reduction% (NO)
productId -> paidPrice (NO)
productId -> whenMade (NO)

Database design - Logical Design

- Example
- Shopping(productId, productName, listPrice, buyerName, reduction%, paidPrice, whenMade)

productName -> productId (perhaps, No)
buyerName -> productId (NO)
buyerName -> reduction% (Maybe, OK)
listPrice, reduction% -> paidPrice (OK)

Database design - Logical Design

- Example
- Shopping(productId, productName, listPrice, buyerName, reduction%, paidPrice, whenMade)
- ProductID , WhenMade and buyerName together determine all attributes and form the key, there are no other keys
- Shopping is not in BCNF (many dependencies violate the rule)

Database design - Logical Design

- How to form relations of BCNF
- 1. Define the functional dependencies, eliminate derivable dependencies
- 2. Define the keys of the relation
- 3. Group the dependencies by the common determinant (left hand side, in X->Y X is determinant)
- 4. Form a relation for each group, include all the attributes in the dependencies of the group

Database design - Logical Design

- 5. If the key of the original relation is not included in any of the relations make a new relation for it.
- 6. If some information is expressed redundantly eliminate this.
- 7. Define names for the schemas. If it's easy to find descriptive names for relations your solution is good.

Database design - Logical Design

- Example
- Shopping(productId, productName, listPrice, buyerName, reduction%, paidPrice, whenMade)

productId -> productName
productId -> listPrice
==> (productId, productName, listPrice) (4)
buyerName -> reduction%
==> (buyerName, reduction%) (4)
listPrice, reduction% -> paidPrice
==>(listPrice, reduction%, paidPrice) (4)

Database design - Logical Design

- Example
- Shopping(productId, productName, listPrice, buyerName, reduction%, paidPrice, whenMade)
- Key is not included
- ==> (productId, buyerName, whenMade)

Database design - Logical Design

- Product(<u>productId</u>, productName, listPrice)
- Customer(<u>buyerName</u>, reduction%)
- MayBeComputed(listPrice, reduction%, paidPrice)
  - may be computed, need not be stored in database
- Shopping(<u>productId, buyerName, whenMade</u>)

Database design - Logical Design

- **In analysing an order form we found the following attributes:**
- form_number,
- who_ordered_id,
- who_ordered_name,
- who_ordered_address,
- who_ordered_phone,
- delivery_address,
- row_no,
- product_code,
- product_name,
- amount_ordered, and
- date_ordered.

Database design - Logical Design

- form_number → who_ordered_id
- who_ordered_id→ who_ordered_name
- who_ordered_id → who_ordered_address
- who_ordered_id → who_ordered_address
- form_number → delivery_address

Database design - Logical Design

- Product_code → product_name
- form_number, row_no → product_code
- form_number, row_no → amount_ordered
- form_number → date_ordered

Database design - Logical Design

- relations

- X(who_ordered_id,who_ordered_name, who_ordered_address, who_ordered_phone)
- Y(product_code, product_name)
- Z(form_number,date_ordered,who_ordered_id)
- T(form_number,row_no,product_code, amount_ordered)

Database design - Logical Design

- Renamed
  - **Customer**(who_ordered_id,who_ordered_name, who_ordered_address, who_ordered_phone)
  - **Product** (product_code, product_name)
  - **Order**(form_number,date_ordered,who_ordered_id)
  - **OrderItem**(form_number,row_no,product_code, amount_ordered)