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.

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 the course Introduction to Application Design and Analysis.

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.

Logical design defines the database for some type of database management system, for example, for a relational dbms.

It considers how the data are represented using the structures offered by the dbms:
- what datatypes to use
- how to organize the data into tables
- what are the keys
- how to connect rows.

If we had done the conceptual design using the E-R or the UML technique, there is a straightforward 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 many times causes problems:
  - storage space is wasted
  - updating of 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

  Location must be repeated if O.East is deleted, also information about Production dept is lost.
  If Admin dept. Moves to Espoo, we must update many roles.

Database design - Logical Design

- We get rid of the problems with tables:

  Employee
  id  eNo  eName     eDate Dept
  10  M.Smith 1.3.59 3
  20  D.Lowe 4.10.40 3
  30  K.Knuth 30.1.66 4
  40  B.West 2.5.65  4
  50  O.East 10.2.55 6

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

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.
According to this table instance it seems that
A→B, D→A, D→B, D→C, AC→D, BC→A

- 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

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

<table>
<thead>
<tr>
<th>Job</th>
<th>Salary</th>
<th>Address</th>
<th>EmpNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>clerk</td>
<td>2000</td>
<td>ccc</td>
<td>10</td>
</tr>
<tr>
<td>clerk</td>
<td>2000</td>
<td>cca</td>
<td>20</td>
</tr>
<tr>
<td>analyst</td>
<td>3000</td>
<td>ccd</td>
<td>30</td>
</tr>
<tr>
<td>analyst</td>
<td>3000</td>
<td>cca</td>
<td>40</td>
</tr>
<tr>
<td>analyst</td>
<td>3000</td>
<td>ccc</td>
<td>50</td>
</tr>
</tbody>
</table>

- 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

There are also other functional dependencies like
- EmpNo, Salary, Address→EmpNo

OK: EmpNo→Job, EmpNo→Salary, EmpNo→Address
OK?: Job→Salary
Not always: Salary, Address→Job
Not always: Job, Address→ EmpNo

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

- Keys and functional 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
    \[ K \rightarrow X \]
    for each attribute \( X \) in \( R \) and no subset of \( K \) has this same property.
  - Thus the key for relation
    \[ \text{Emp(Job,Salary,Address,EmpNo)} \]
    is \( \text{EmpNo} \)

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 functional dependencies \( X \rightarrow Y \) related to it such that \( X \) does not contain a key of the relation
- \( \text{Emp(Job,Salary,Address,EmpNo)} \) is not in BCNF because its key is \( \text{EmpNo} \) and there is the dependency \( \text{Job} \rightarrow \text{Salary} \), where \( \text{EmpNo} \) is not part of \( \text{Job} \).

Example

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

- \( \text{productId} \rightarrow \text{productName} \) (OK)
- \( \text{productId} \rightarrow \text{listPrice} \) (OK)
- \( \text{productId} \rightarrow \text{buyerName} \) (NO) {only one buyer for each product}
- \( \text{productId} \rightarrow \text{reduction%} \) (NO) {reduction is product specific}
- \( \text{productId} \rightarrow \text{paidPrice} \) (NO) {paid price depends on product only}
- \( \text{productId} \rightarrow \text{whenMade} \) (NO) {only one shopping for a product}

Example

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

- \( \text{productName} \rightarrow \text{productId} \) (perhaps, No)
- \( \text{buyerName} \rightarrow \text{productId} \) (NO) {nobody buys more than one product}
- \( \text{buyerName} \rightarrow \text{reduction%} \) (Maybe, OK)
- \( \text{listPrice, reduction%} \rightarrow \text{paidPrice} \) (OK)

Example

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

- \( \text{ProductID} , \text{WhenMade} \) and \( \text{buyerName} \) together determine all attributes and form the key, there are no other keys
  - (note: attributes that are not determined by any other attributes must be included in all keys)
- Shopping is not in BCNF (many dependencies violate the rule)

Example

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

- 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 \( X \rightarrow Y \) attribute \( X \) is the determinant)
  4. Form a relation for each group, include in the schema all the attributes in the dependencies of the group
5. If no key of the original relation is included in any of the relations make a new relation for one of the keys.

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.

**Example**

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

```
productId -> productName
productId -> listPrice
buyerName -> reduction%
listPrice, reduction% -> paidPrice
```

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

**Example**

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

Key was not included

```
==> (productId, buyerName, whenMade)
```

**Example**

```
Product(productId, productName, listPrice)
Customer(buyerName, reduction%)
MayBeComputed(listPrice, reduction%, paidPrice)
```

– may be computed, need not be stored in database

```
Shopping(productId, buyerName, whenMade)
```

**Example**

```
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.
```

**Example**

```
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 $\rightarrow$ product_name
- form_number, row_no $\rightarrow$ product_code
- form_number, row_no $\rightarrow$ amount_ordered
- form_number $\rightarrow$ 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, delivery_address)
  T(form_number, row_no, product_code, amount_ordered)

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, delivery_address)
- OrderItem(form_number, row_no, product_code, amount_ordered)