Applied Databases

Handout 2a. Functional Dependencies and Normal Forms

20 Oct 2008

Functional Dependencies

This is the most "mathematical" part of the course. Functional dependencies provide an alternative approach to database design.

Why you need to understand them:

1. They are a mathematical, rigourous formulation.
2. They are good for checking database design and anomalies in database design.

Why you don't want to understand them:

1. They are a mathematical, rigourous formulation.
2. The approach is incomplete and, if extended, gets far too instense.

Not all designs are equally good!

- Why is this design bad?

```
Data(Id, Name, Address, CId, Description, Grade)
```

- And why is this design good?

```
Student(Id, Name, Address)
Course(CId, Description)
Enrolled(Id, CId, Grade)
```


An example of the "bad" design

Id	Name	Address	CId	Description	Grade
124	Knox	Troon	Phil2	Plato	A
234	McKay	Skye	Phil2	Plato	B
789	Brown	Arran	Math2	Topology	C
124	Knox	Troon	Math2	Topology	A
789	Brown	Arran	Eng3	Chaucer	B

- Some information is *redundant*, e.g. `Name` and `Address`.
- Without null values, some information cannot be represented, e.g, a student taking no courses.

Functional Dependencies

- Recall that a *key* is a set of attribute names. If two tuples agree on the a key, they agree everywhere (they are the same).
- In our "bad" design, `Id` is not a key, but if two tuples agree on `Id` then they agree on `Address`, even though the tuples may be different.
- We say "`Id` *determines* `Address`" written `Id` $\rightarrow$ `Address`.
- A functional dependency is a *constraint* on instances.

Example

Here are some functional dependencies that we expect to hold in our student-course database:

`Id` $\rightarrow$ `Name, Address`
`CId` $\rightarrow$ `Description`
`Id, CId` $\rightarrow$ `Grade`

Note that an instance of any schema (good or bad) should be constrained by these dependencies.

A functional dependency $X \rightarrow Y$ is simply a pair of sets. We often use sloppy notation $A, B \rightarrow C, D$ or $AB \rightarrow CD$ when we mean $\{A, B\} \rightarrow \{C, D\}$

Functional dependencies (fd's) are integrity constraints that subsume keys.

Definition

Def. Given a set of attributes R, and subsets X, Y of R, an instance $r$of R satisfies the functional dependency $X \longrightarrow Y$ if for any tuples t_1, t_2 in r, whenever $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$.

(We use $t[X]$ to mean the "projection" of the tuple t on attributes X)

We say "X functionally determines Y" or "X determines Y"

A *superkey* (a superset of a key) is simply a set X such that $X \rightarrow R$

A *key* can now be defined, somewhat perversely, as a minimimal superkey.

The Basic Intuition in Relational Design

A database design is "good" if all fd's are of the form $K \rightarrow R$, where K is a key for R.

Example: our bad design is bad because `Id` $\rightarrow$ `Address`, but `Id` is not a key for the table.

But it's not quite this simple. $A \rightarrow A$ always holds, but we don't expect any attribute A to be a key!

There are lots of functional dependencies!

Functional dependencies generate other functional dependencies, using "Armstrong's Axioms":

1. *Reflexivity*: if $Y \subseteq X$ then $X \to Y$
   (These are called **trivial** dependencies.)
   Example: `Name, Address` $\to$ `Address`
2. *Augmentation*: if $X \to Y$ then $X \cup W \to Y \cup W$
   Example: Given `CId` $\to$ `Description`, then `CId,Id` $\to$ `Description,Id`. Also, `CId` $\to$ `Description,CId`
3. *Transitivity*: if $X \to Y$ and $Y \to Z$ then $X \to Z$
   Example: Given `Id,CId` $\to$ `CId` and `CId` $\to$ `Description`, then `Id, CId` $\to$ `Description`

Consequences of Armstrong's Axioms

1. Union: if $X \to Y$ and $X \to Z$ then $X \to Y \cup Z$.
2. Pseudotransitivity: if $X \to Y$ and $W \cup Y \to Z$ then $X \cup W \to Z$.
3. Decomposition: if $X \to Y$ and $Z \subseteq Y$ then $X \to Z$

Try to prove these using Armstrong's Axioms!

An example

Proof of union.

1. $X \to Y$ and $X \to Z$ [Assumption]
2. $X \to X \cup Y$ [Assumption and augmentation]
3. $X \cup Y \to Z \cup Y$ [Assumption and augmentation]
4. $X \to Y \cup Z$ [2, 3 and transitivity]

Closure of an fd set

Def. The closure F^+ of an fd set F is given by

$$\{X \to Y \mid X \to Y \text{ can be deduced from } F \text{ Armstrong's axioms}\}$$

Def. Two fd sets F, G are equvalent if $F^+ = G^+$.

Unfortunately, the closure of an fd set is huge (how big?) so this is not a good way to test whether two fd sets are equivalent.

A better way is to test whether each fd in one set follows from the other fd set and *vice versa*.

Closure of an attribute set

Given a fd set set F, the closure X^+ of an attribute set X is given by:

$$X^+ = \bigcup\{Y \mid X \to Y \in F^+\}$$

Example. What are the the following?

- $\{$`Id`$\}^+$
- $\{$`Id,Address`$\}^+$
- $\{$`Id,CId`$\}^+$
- $\{$`Id,Grade`$\}^+$

Implication of a fd

"Is $X \to Y \in F^+$?" ("Is $X \to Y$ implied by the fd set F") can be answered by checking whether Y is a subset of X^+. X^+ can be computed as follows:

```
X+ := X
while there is a fd U → V in F such that U ⊆ X+ and V ⊈ X+
    X+ := X+ ∪ V
```

Try this with `Id,CId` → `Description,Grade`

The general goal, to repeat

No "embedded" functional dependencies. For example the table `(Id, Name, CId)` is not a good design, because {`Id,CId`} is the key; `Id` alone is not a key.

Why don't we decompose into, say, {`Id, Name, Address, Grade`} and {`CId, Description`}?

Or into {`Id, Name, Address`}, {`CId, Description`} and {`Grade`}?

We need some conditions on decomposition.

Lossless join decomposition

$R_1, R_2, \ldots, R_k$ is a *lossless join* decomposition with respect to a fd set F if, for every intance of R that satisfies F,

$$\pi_{R_1}(r) \bowtie \pi_{R_2}(r) \ldots \pi_{R_k}(r) = r$$

Example:

Id	Name	Address	CId	Description	Grade
124	Knox	Troon	Phil2	Plato	A
234	McKay	Skye	Phil2	Plato	B

What happens if we decompose on {`Id, Name, Address`} and {`CId, Description, Grade`} or on {`Id, Name, Address, Description, Grade`} and {`CId, Description`}?

Testing for a lossless join

Fact. R_1, R_2 is a lossless join decomposition of R with respect to F if at least one of the following dependencies is in F^+:

$$(R_1 \cap R_2) \rightarrow R_1 - R_2$$
$$(R_1 \cap R_2) \rightarrow R_2 - R_1$$

Example: with respect to the fd set

```
     Id  →  Name, Address
    CId  →  Description
Id, CId  →  Grade
```

is {Id, Name, Address} and {Id, CId, Description, Grade} a lossless decomposition?

Dependency Preservation

Given a fd set F, we'd like a decomposition to "preserve" F. Roughly speaking we want each $X \rightarrow Y$ in F to be contained within one of the attribute sets of our decomposition.

Def. The *projection* of an fd set F onto a set of attributes Z, F_Z is given by:

$$F_Z = \{X \rightarrow Y \ \mid \ X \rightarrow Y \in F^+ \text{ and } X \cup Y \subseteq Z\}$$

A decomposition $R_1, R_2, \ldots R_k$ is *dependency preserving* if

$$F^+ = (F_{R_1} \cup F_{R_2} \cup \ldots \cup F_{R_k})^+$$

If a decomposition is dependency preserving, then we can easily check that an update on an instance R_i does not violate F by just checking that it doesn't violate those fd's in F_{R_i}.

Example 1

The scheme: {Class, Time, Room}

The fd set:
```
    Class  →  Room
Room,Time  →  Class
```

The decomposition: {Class, Room} and {Room, Time}

Is it *lossless*?

Is it *dependency preserving?*

Example 2

The scheme: {Student, Time, Room, Course, Grade}

The fd set:
```
  Student, Time  →  Room
Student, Course  →  Grade
```

The decomposition: {Student, Time, Room} and {Student, Course, Grade}

It it lossless?

Is it dependency preserving?

Relational Database Design

Earlier we stated that the idea in analysing fd sets is to find a design (a decomposition) such that for each non-trivial dependency $X \to Y$ (non-trivial means $Y \not\subseteq X$), X is a superkey for some relation scheme in our decomposition.

Example 1 shows that it is not possible to achieve this and to preserve dependencies.

This leads to two notions of normal forms....

Normal forms

Boyce-Codd Normal Form (BCNF) For every relation scheme R in the decomposition, and for every $X \to A$ that holds on R (that is, $X \cup \{A\} \subseteq R$, either

- $A \in X$ (it is trivial), or
- X is a superkey for R.

Third Normal Form (3NF) For every relation scheme R and for every $X \to A$ that holds on R,

- $A \in X$ (it is trivial), or
- X is a superkey for R, or
- A is a member of some key of R (A is "prime")

Observations on Normal Forms

BCNF is stronger than 3NF.

BCNF is clearly desirable, but example 1 shows that it is not always achievable.

There are algorithms to obtain

- a BCNF lossless join decomposition
- a 3NF lossless join, dependency preserving decomposition

So what's this all for?

Even though there are algorithms for designing databases this way, they are hardly ever used. People normally use E-R diagrams and the like. But...

- Automated procedures (or human procedures) for generating relational schemas from diagrams often mess up. Further decomposition is sometimes needed (or sometimes thet decompose too much, so merging is needed)
- Understanding fd's is a good "sanity check" on your design.
- It's important to have these criteria. Bad design w.r.t. these criteria often means that there is redundancy or loss of information.
- For efficiency we sometimes design redundant schemes deliberately. Fd analysis allows us to identify the redundancy.

Functional dependencies – review

- Redundancy and update anomalies.
- Functional dependencies.
- Implication of fd's and Armstrong's axioms.
- Closure and equivalence of fd sets.
- Lossless join decomposition and dependency preservation.
- BCNF and 3NF.