DBS Database Systems
Designing Relational Databases

Peter Buneman

12 October 2010

SQL DDL

In its simplest use, SQL's *Data Definition Language* (DDL) provides a name and a type for each column of a table.

```
CREATE TABLE Hikers (  HId     INTEGER,
                       HName   CHAR(40),
                       Skill   CHAR(3),
                       Age     INTEGER )
```

In addition to describing the type or a table, the DDL also allows you to impose constraints. We'll deal with two kinds of constraints here: *key constraints* and *inclusion constraints*

Key Constraints

A *key* is a subset of the attributes that constrains the possible instances of a table. For any instance, no two distinct tuples can agree on their key values.

Any superset of a key is also a key, so we normally consider only minimal keys.

```
CREATE TABLE Hikers (   HId     INTEGER,
                        HName   CHAR(30),
                        Skill   CHAR(3),
                        Age     INTEGER,
                        PRIMARY KEY (HId)        )

CREATE TABLE Climbs (   HId     INTEGER,
                        MId     INTEGER,
                        Date    DATE,
                        Time    INTEGER,
                        PRIMARY KEY (HId, MId)  )
```

Updates that violate key constraints are rejected.

Do you think the key in the second example is the right choice?

Inclusion Constraints

A field in one table may refer to a tuple in another relation by indicating its key. The referenced tuple must exist in the other relation for the database instance to be valid. For example, we expect any `MId` value in the `Climbs` table to be included in the `MId` column of the `Munros` table.

SQL provides a restricted form of inclusion constraint, *foreign key* constraints.

```
CREATE TABLE Climbs (  HId    INTEGER,
                       MId    INTEGER,
                       Date   DATE,
                       Time   INTEGER,
                       PRIMARY KEY (HId, MId),
                       FOREIGN KEY (HId) REFERENCES Hikers(HId),
                       FOREIGN KEY (MId) REFERENCES Munros(MId)  )
```

There's much more to SQL DDL

Cardinality constraints, triggers, views. There are also many features for controlling the *physical* design of the database.

Some of these will appear later in the course.

However, the two simple constraints that we have just seen, key constraints and foreign key constraints are the basis for database design.

SQL – Summary

SQL extends relational algebra in a number of useful ways: arithmetic, multisets as well as sets, aggregate functions, group-by. It also has updates both to the data and to the schema. "Embeddings" exist for many programming languages. However, there are a number of things that cannot be expressed in SQL:

- Queries over *ordered structures* such as lists.
- Recursive queries.
- Queries that involve nested structures (tables whose entries are other tables)

Moreover SQL is not extensible. One cannot add a new base type, one cannot add new functions (e.g., a new arithmetic or a new aggregate function)

Some of these limitations are lifted in query languages for object-relational and object-oriented systems.

Conceptual Modelling and Entity-Relationship Diagrams

[R&G Chapter 2]

Obtaining a good database design is one of the most challenging parts of building a database system. The database design specifies what the users will find in the database and how they will be able to use it.

For simple databases, the task is usually trivial, but for complex databases required that serve a commercial enterprise or a scientific discipline, the task can daunting. One can find databases with 1000 tables in them!

A commonly used tool to design databases is the *Entity Relationship* (E-R) model. The basic idea is simple: to "conceptualize" the database by means of a diagram and then to translate that diagram into a formal database specification (e.g. SQL DDL)

Conceptual Modelling – a Caution

There are many tools for conceptual modelling some of them (UML, Rational Rose, etc.) are designed for the more general task of software specification. E-R diagrams are a subclass of these, intended specifically for databases. They all have the same flavour.

Even within E-R diagrams, no two textbooks will agree on the details. We'll follow R&G, but be warned that other texts will use different convernmtions (especially in the way many-one and many-many relationships are described.)

Unless you have a formal/mathematical grasp of the meaning of a diagram, conceptual modelling is almost guaranteed to end in flawed designs.

Conceptual Design

- What are the *entities* and *relationships* that we want to describe?
- What information about entities and relationships should we store in the database?
- What *integrity constraints* hold?
- Represent this information pictorially in an E-R diagram, then map this diagram into a relational schema (SQL DDL.)

ER digarams – the basics

In ER diagrams we break the world down into three kinds of things:

- Attributes. These are the things that we typically use as column names: Name, Age, Height, Address etc.

  Attributes are drawn as ovals: **Name**
- Entities. These are the real world "objects" that we want to represent: Students, Courses, Munros, Hikers, ... A database typically contains sets of entitites.

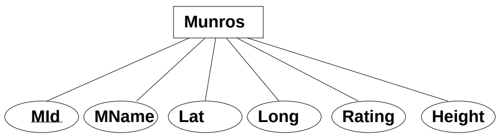
  Entity sets are drawn as boxes: **Courses**
- Relationships. This describes relationships among entitites, e.g. a students *enrolls* in a course, a hiker *climbs* a Munro, ...

  Relationships are drawn as diamonds: **Enrolls**

Drawing Entity Sets

The terms "entity" and "entity set" are often confused. Remember that boxes describe *sets* of entities.

To draw an entity set we simpy connect it with its attributes

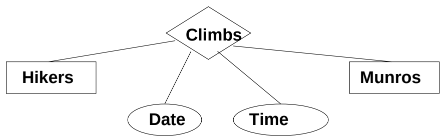


Note that we have indicated the key for this entity set by underlining it.

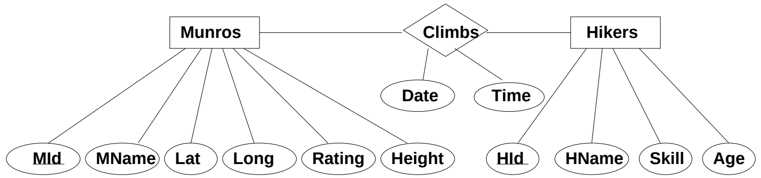
Drawing Relatonships

We connect relationships to the entities they "relate". However relationships can also have attributes. Note that `Date` and `Time` apply to `Climbs` – not to `Hikers` or `Munros`.

We connect relationships to enitity sets and attributes in the same way that we connected entity sets to attributes.



The whole diagram



Note that lines connect entitites to attributes and relationships to entities and to attributes. They do *not* connect attributes to attributes, entitites to entities, or relationships to relationships. This is a "toy" diagram. Real ER diagrams can cover a whole wall or occupy a whole book!

Obtaining the relational schema from an ER diagram

We now translate the ER diagram into a relational schema. Roughly epaking (this will not always be the case) we generate a table for each entity and a table each relationship.

For each entity we generate a relation with the key that is specified in the ER diagram. For example (SQL DDL)

```
CREATE TABLE Munros (
     MId      INTEGER,
     MName    CHAR(30),
     Lat      REAL,
     Long     REAL,
     Height   INTEGER,
     Rating   REAL,
     PRIMARY KEY (RId)  )
```

```
CREATE TABLE Hikers (
     HId     INTEGER,
     HName   CHAR(30),
     Skill   CHAR(3),
     Age     INTEGER,
     PRIMARY KEY (HId)  )
```


Obtaining the relational schema – continued

For each relationship we generate a relation scheme with attributes

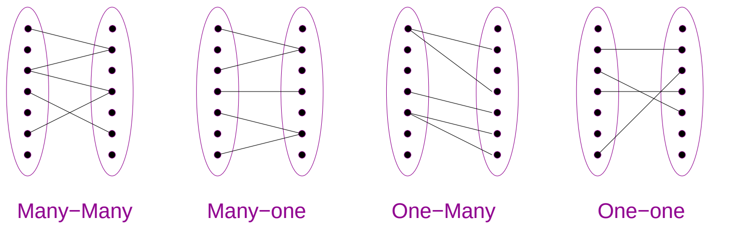
- The key(s) of each associated entity
- Additional attribute keys, if they exist
- The associated attributes.

Also, the keys of associated attributes are foreign keys.

```
CREATE TABLE Climbs (   HId    INTEGER,
                        MId    INTEGER,
                        Date   DATE,
                        Time   REAL,
                        PRIMARY KEY (HId,MId), ← also Date?
                        FOREIGN KEY (HId) REFERENCES Hikers,
                        FOREIGN KEY (MId) REFERENCES Munros );
```

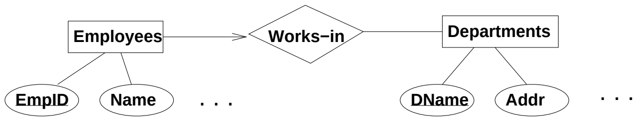

Many-one Relationships

The relationship `Climbs` represents – among other things – a relation (in the mathematical sense) between the sets associated with `Munros` and `Hikers`. That is, a subset of the set of Munro/Hiker pairs. This is a *many-many* relation, but we need to consider others.



A Many-one relationship

Consider the relationship between `Employees` and `Departments`. An Employee works in at most one department. There is a *many-one* relationship between `Employees` and `Departments` indicated by an arrow emanating from `Employees`



Note that an employee can exist without being in a department, and a department need not have any employees.

The Associated DDL

```
CREATE TABLE Departments (
   DeptID    INTEGER,
   Address   CHAR(80),
   PRIMARY KEY (DeptId) )
```

and

```
CREATE TABLE Employees (
    EmpID    INTEGER,
    NAME     CHAR(10)
    PRIMARY KEY (EmpId) )

CREATE TABLE WorksIn (
    EmpID    INTEGER,
    DeptID   INTEGER,
    PRIMARY KEY (EmpId),
    FOREIGN KEY (EmpId)
      REFERENCES Employees,
    FOREIGN KEY (DeptID)
      REFERENCES Departments )
```

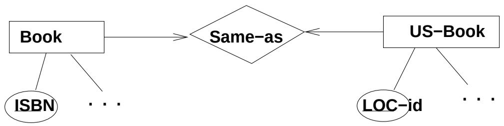
or

```
CREATE TABLE Employees (
   EmpID    INTEGER,
   NAME     CHAR(10),
   DeptID   INTEGER,
   PRIMARY KEY (EmpId),
   FOREIGN KEY DeptID
     REFERENCES Departments)
```

The key for `WorksIn` has "migrated" to `Employees`.

1 – 1 Relationships?

These are typically created by database "fusion". They arise through various "authorities" introducing their own identification schemes.

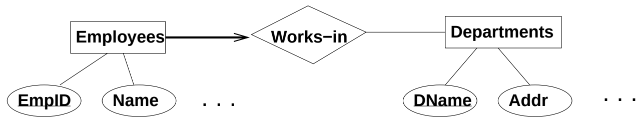


The problem is that such a relationship is never quite 1-1. E.g. *Scientific Name* and PubMed identifiers for taxa.

When can one "migrate" a key?

Participation Constraints

Suppose we also want to assert that every employee must work in some department. This is indicated (R&G convention) by a *thick* line.




```
CREATE TABLE Departments (
   DeptID    INTEGER,
   Address   CHAR(80),
   PRIMARY KEY (DeptId) )
```

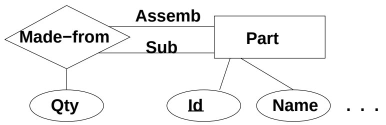
and

```
CREATE TABLE Employees (
   EmpID    INTEGER,
   NAME     CHAR(10),
   DeptID   INTEGER NOT NULL,
   PRIMARY KEY (EMPID),
   FOREIGN KEY DeptID
     REFERENCES Departments )
```

Note: Many-one = *partial function*, many-one + participation = *total function*

Labelled Edges

It can happen that we need two edges connecting an entity set with (the same) relationship.

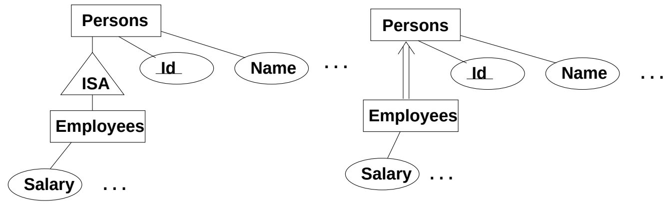


When one sees a figure like this there is typically a recursive query associated with it, e.g., "List all the parts needed to make a widget."

What are the key and foreign keys for `Made-from`?

ISA relationships

An *isa* relationship indicates that one entity is a "special kind" of another entity.



The textbook draws this relationship as shown on the left, but the right-hand representation is also common.

This is not the same as o-o inheritance. Whether there is inheritance of methods depends on the representation and the quirks of the DBMS. Also note that, we expect some form of *inclusion* to hold between the two entity sets.

Relational schemas for ISA

```
CREATE TABLE Persons (
   Id     INTEGER,
   Name   CHAR(22),
   ...
   PRIMARY KEY (Id) )
```

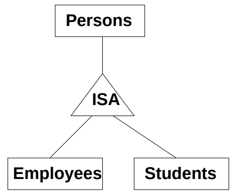
```
CREATE TABLE Employees (
      Id       INTEGER,
      Salary   INTEGER,
...
      PRIMARY KEY (Id),
      FOREIGN KEY (Id) REFERENCES Persons )
```

A problem with this representation is that we have to do a join whenever we want to do almost any interesting query on `Employees`.

An alternative would be to have all the attributes of `Persons` in a *disjoint* `Employees` table. What is the disadvantage of this representation? Are there other representations?

Disjointness in ISA relationships

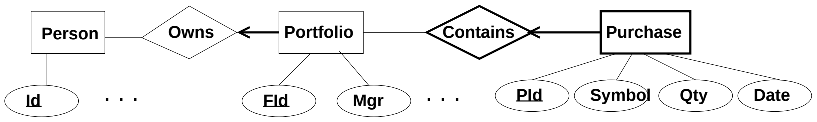
When we have two entities that are both subclasses of some common entity it is always important to know whether they should be allowed to overlap.



Can a person be both a student and an employee? There are no mechanisms in SQL DDL for requiring the two sets to be exclusive. However it is common to want this constraint and it has to be enforced in the applications that update the database.

Weak Entities

An entity that depends on another entity for its existence is called a *weak entity*.



In this example a `Purchase` cannot exist unless it is in a `Portfolio`. The key for a `Purchase` may be a compound `FId`/`PId`. Weak entities are indicated in R&G by thick lines round the entity and relationship.

Weak entities tend to show up in XML design. The hierarchical structure limits what we can do with data models.

Weak Entities – the DDL

```
CREATE TABLE Portfolio (
   FId      INTEGER,
   Owner    INTEGER,
   Mgr      CHAR(30),
   PRIMARY KEY (FId),
   FOREIGN KEY (Owner)
     REFERENCES Person(Id) )
```

```
CREATE TABLE Purchase (
   PId      INTEGER,
   FId      INTEGER,
   Symbol   CHAR(5),
   QTY      INTEGER,
   Date     DATE
   PRIMARY KEY (FId, PId),
   FOREIGN KEY (FId)
      REFERENCES Portfolio
      ON DELETE CASCADE )
```

`ON DELETE CASCADE` means that if we delete a portfolio, all the dependent `Purchase` tuples will automatically be deleted.

If we do not give this incantation, we will not be able to delete a portfolio unless it is "empty".

Other stuff you may find in E-R diagrams

- Cardinality constraints, e.g., a student can enroll in at most 4 courses.
- Aggregation – the need to "entitise" a relationship.
- Ternary or n-ary relationships. No problem here, but our diagrams aren't rich enough properly to extend the notion of many-one relationships.

It is very easy to go overboard in adding arbitrary features to E-R diagrams. Translating them into types/constraints is another matter. Semantic networks from AI had the same disease – one that is unfortunately re-infecting XML.

E-R Diagrams, Summary

E-R diagrams and related techniques are the most useful tools we have for database design.

The tools tend to get over-complicated, and the complexities don't match the types/constraint systems we have in DBMSs

There is no agreement on notation and little agreement on what "basic" E-R diagrams should contain.

The semantics of E-R diagrams is seldom properly formalized. This can lead to a lot of confusion.

Review

- Basics, many-one, many-many, etc.
- Maping to DDL
- "Entitising" relationships.
- Participation, ISA, weak entities.

Relational Database Design and Functional Dependencies

Reading: R&G Chapter 19

- We don't use this to *design* databases (despite claims to the contrary.)
- ER-diagrams are much more widely used.
- The theory is useful
  - as a check on our designs,
  - to understand certain things that ER diagrams *cannot* do, and
  - to help understand the consequences of redundancy (which we may use for efficiency.)
  - also in OLAP designs and in data cleaning

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

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

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

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

Definition

Def. Given a set of attributes R, and subsets X, Y of R, $X \longrightarrow Y$ is a *functional dependency* (read "X functionally determines Y" or "X determines Y") if for any instance r of R, and 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)

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

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

The Basic Intuition in Relational Design

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

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

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

Armstrong's Axioms

Functional dependencies have certain consequences, which can be reasoned about using Armstrong's Axioms:

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

Consequences of Armstrong's Axioms

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

Try to prove these using Armstrong's Axioms!

An example

Proof of union.

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

Closure of an fd set

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

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

Def. Two fd sets F, G are equivalent 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` $\to$ `Description,Grade`

Minimal Cover

A set of functional dependencies F is a *minimal cover* iff

1. Every functional dependency in F is of the form $X \to A$ where A is a single attribute.
2. For no $X \to A$ in F is $F - \{X \to A\}$ equivalent to F
3. For no $X \to A$ in F and $Y \subset X$ is $F - \{X \to A\} \cup \{Y \to A\}$ equivalent to F

Example: $\{A \to C, A \to B\}$ is a minimal cover for $\{AB \to C, A \to B\}$

A minimal cover *need not be unique*. Consider $\{A \to B, B \to C, C \to A\}$ and $\{A \to C, B \to A, C \to B\}$

Why Armstrong's Axioms?

Why are Armstrong's axioms (or an equivalent rule set) appropriate for fd's.

They are *consistent* and *complete*.

"Consistent" means that any instance that satisfies every fd in F will satisfy every derivable fd – the fd's in F^+

"Complete" means that if an fd $X \to Y$ cannot be derived from F then there is an instance satisfying F but not $X \to Y$.

In other words, Armstrong's axioms derive exactly those fd's that can be expected to hold.

Proof of consistency

This comes directly from the definition. Consider augmentation, for example. This says that if $X \longrightarrow Y$ then $X \cup W \longrightarrow Y \cup W$.

If an instance$I$ satisfies $X \longrightarrow Y$ then, by definition, for any two tuples t_1, t_2 in I, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. If, in adition, $t_1[W] = t_2[W]$ then $t_1[Y \cup W] = t_2[Y \cup W]$.

Go through all the other axioms similarly.

Proof of completeness

We suppose $X \longrightarrow Y \notin F^+$ and construct an instance that satisfies F^+ but not $X \longrightarrow Y$.

We first observe that, since $X \longrightarrow Y \notin F^+$, there is at least one (single) attribute $A \in Y$ such that $X \longrightarrow A \notin F^+$. Now we construct the table with two tuples that agree on X^+ but disagree everywhere else.

X				A	$X^+ - X$				rest of R		
x_1	x_2	...	x_n	$a_{1,1}$	v_1	v_2	...	v_n	$w_{1,1}$	$w_{2,1}$	...
x_1	x_2	...	x_n	$a_{1,2}$	v_1	v_2	...	v_n	$w_{1,2}$	$w_{2,2}$	...

Obviously this table fails to satisfy $X \longrightarrow Y$. We also need to check that it satisfies any fd in F and hence any fd in F^+

Decomposition

Consider the attribute our attribute set. We have agreed that we need to decompose it in order to get a good design, but how?

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

Why is this decomposition bad?

```
R1(Id, Name, Address)
R2(CId, Description,Grade)
```

Information is lost in this decomposition but how do we express this loss of information?

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^+:

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

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 $X \to Y$ 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:

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

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?*

What about the decomposition {`Class, Room`} and {`Class, Time`}?

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 and for every $X \to A$ that holds on 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

The 3NF algorithm uses a minimal cover.

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 of an fd set.
- Minimal cover.
- Lossless join decomposition and dependency preservation.
- BCNF and 3NF.