Applied Databases

Handout 2. Database Design.

5 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 uniquely identifies a tuple, and for which no subset of the key has this property.

```
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)  )
```

Schema modification

Extremely useful, because database requirements change over time. Examples

1. `DROP TABLE Hikers;`
2. `DROP VIEW Mypeaks;`
3. `ALTER TABLE Climbs ADD Weather CHAR(50);`
4. `ALTER TABLE Munros DROP Rating;`

Almost all of these could violate an integrity constraint or cause a "legacy" program to fail.

Only `ALTER TABLE ... ADD ...` is usually innocuous. It is also very useful.

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:

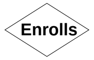
- Attributes. These are the things that we typically use as column names: Name, Age, Height, Address etc.

  Attributes are drawn as ovals: 

- 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: 

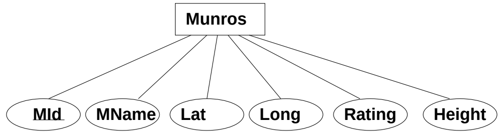
- Relationships. This describes relationships among entitites, e.g. a students *enrolls* in a course, a hiker *climbs* a Munro, ...

  Relationships are drawn as diamonds: 

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

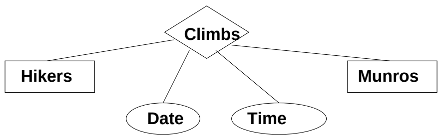


Note that we have indicated the key for this entity set by underlining it.

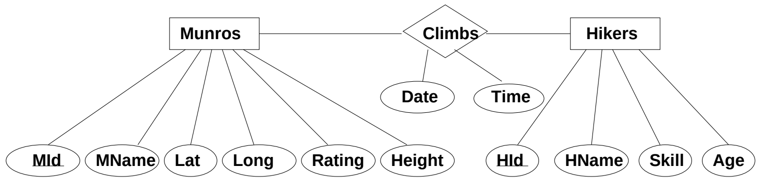
Drawing Relatonships

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. Initially, (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

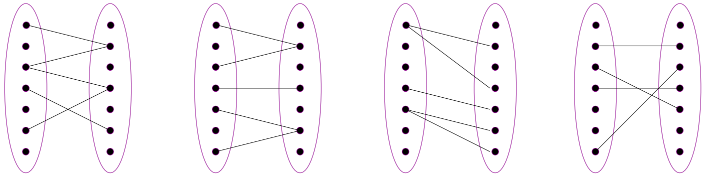
- The key(s) of each associated entity
- Additional attribute keys, if they exist
- The associated attributes.

Also, the keys of associated entities 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

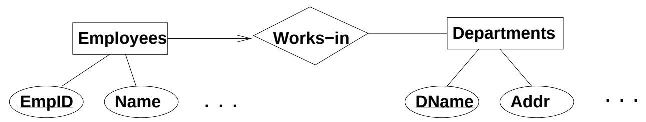
The relationship `Climbs` represents – among other things – a relation (in the mathematical sense) between the sets associated with `Munros` and `Hikers`. That is, a subset of the set of Munro/Hiker pairs. This is a *many-many* relation, but we need to consider others.



Many–Many   Many–one   One–Many   One–one

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 )
```

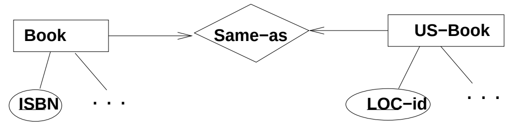
or

```
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?

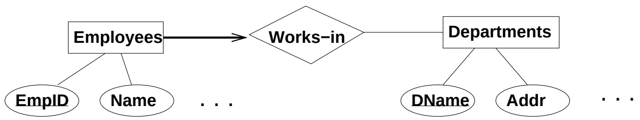
These are uncommon and are typically created by database "fusion".



Why can't one "migrate" a key in this case?

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 (EmpId) )
```

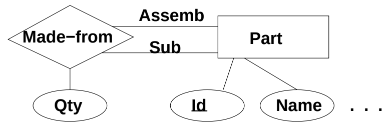
and

```
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.

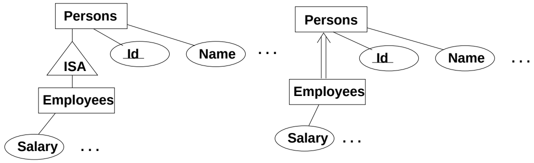


When one sees a figure like this there is typically a recursive query associated with it, e.g., "List all the parts needed to make a widget."

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

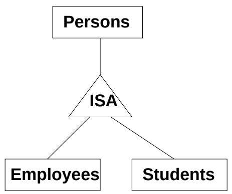
```
CREATE TABLE Persons (
   Id      INTEGER,
   Name    CHAR(22),
   ...
   PRIMARY KEY (Id) )

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

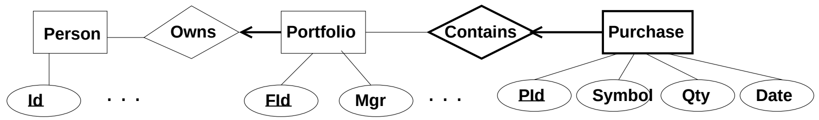
When we have two entities that are both subclasses of some common entity it is always important to know whether they should be allowed to overlap.



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.

Lecture 2, Review

- SQL DDL and schema modification
- E-R diagrams
  - Basics, many-one, many-many, etc.
  - Maping to DDL
  - Participation, ISA, weak entities.