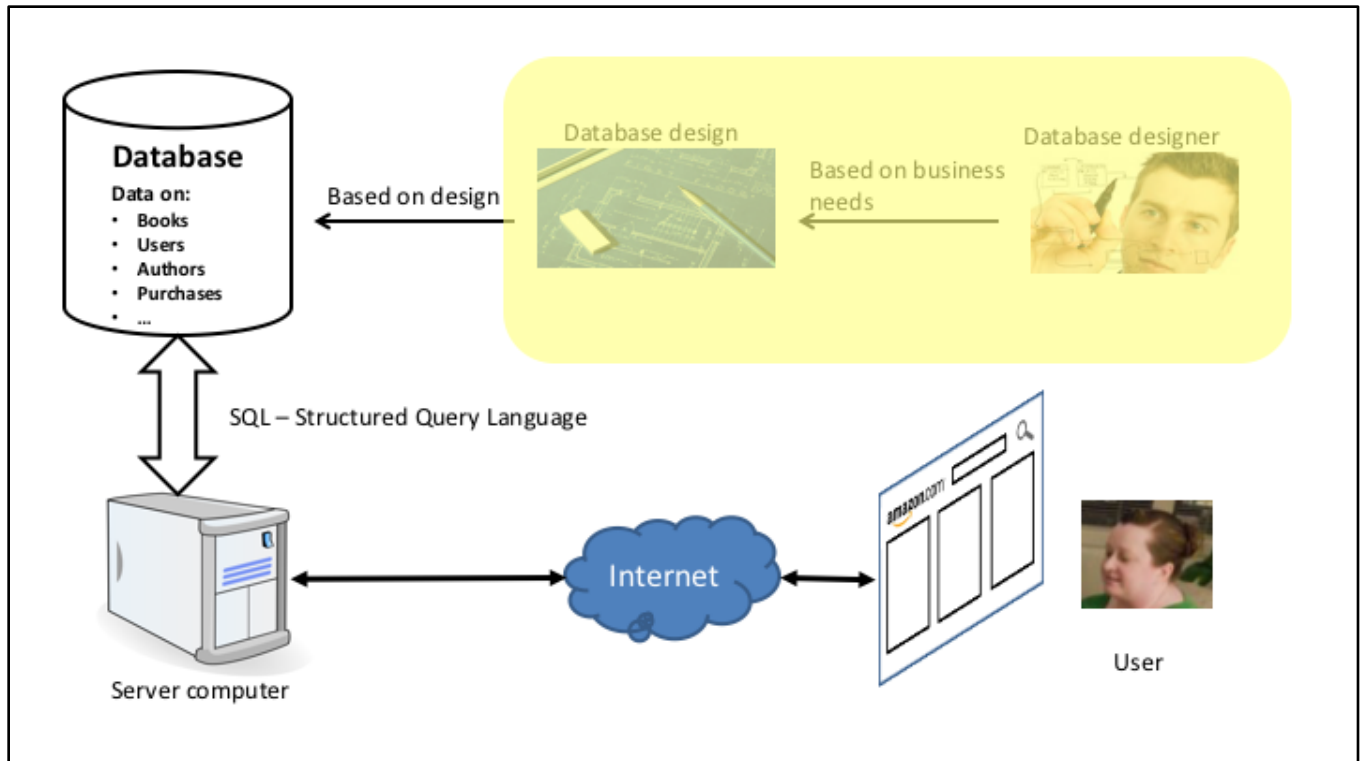


Database Design: Part 4



We have already looked at how SQL can help us to retrieve almost any information we need from a relational database.

We have gone quite deep into database design to some extent.

A few more key concepts (no pun intended), and a look at a few more problems will help us to wrap it up.

Every employee (other than the CEO) reports to one manager and each manager has zero or more subordinates.

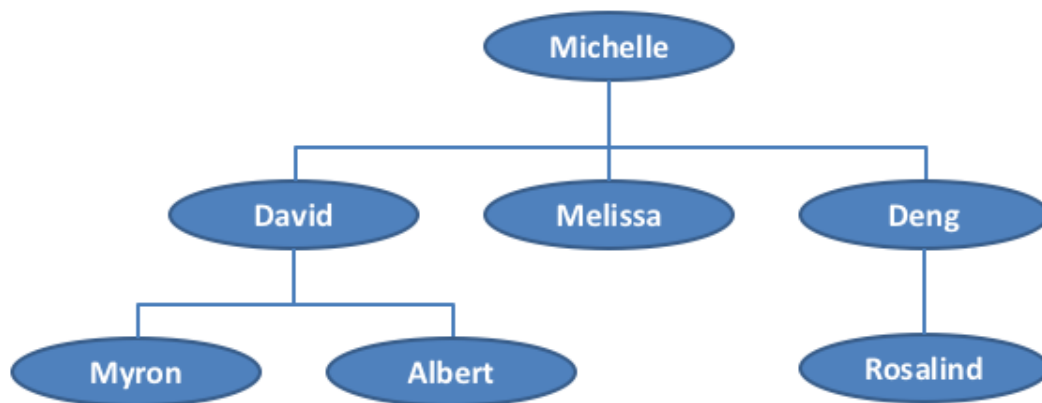


For the above description, we might be tempted to come up with something like the ERD shown.

However, that would be wrong – because managers are also employees! The diagram makes it look as if managers and employees are instances of different entity types.

Most people in an organization are managers and a subordinates at the same time.

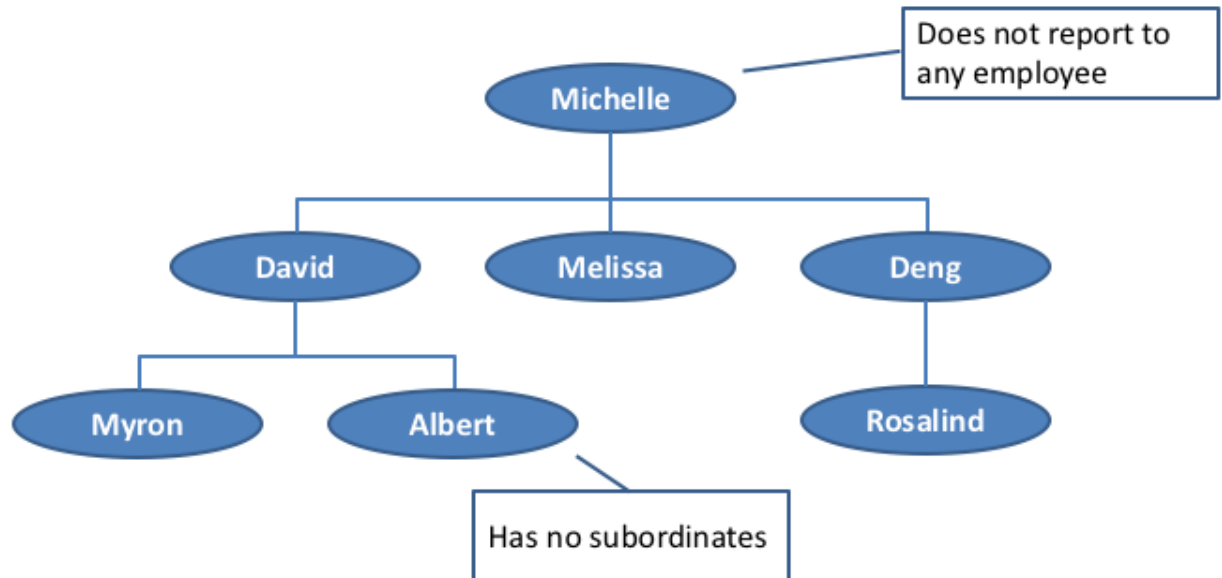
So how do we deal with this?



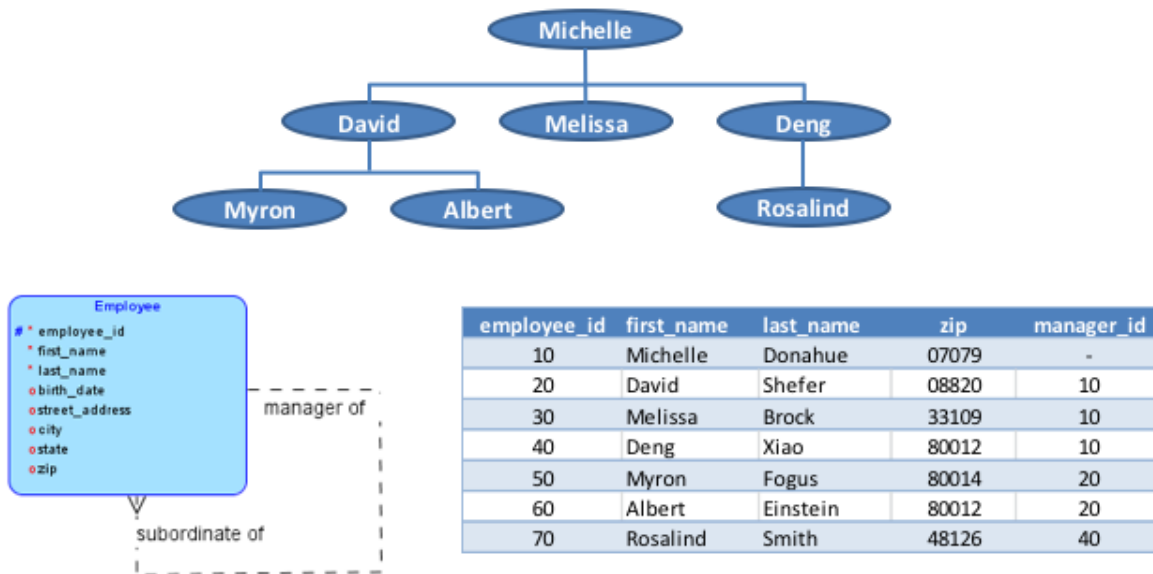
A sample organization chart. The CEO does not report to any other employees and the employees at the bottom level do not have any subordinates. All others are both managers and subordinates.

One employee reports to one other employee or none

One employee manages zero or many employees



Unary relationship



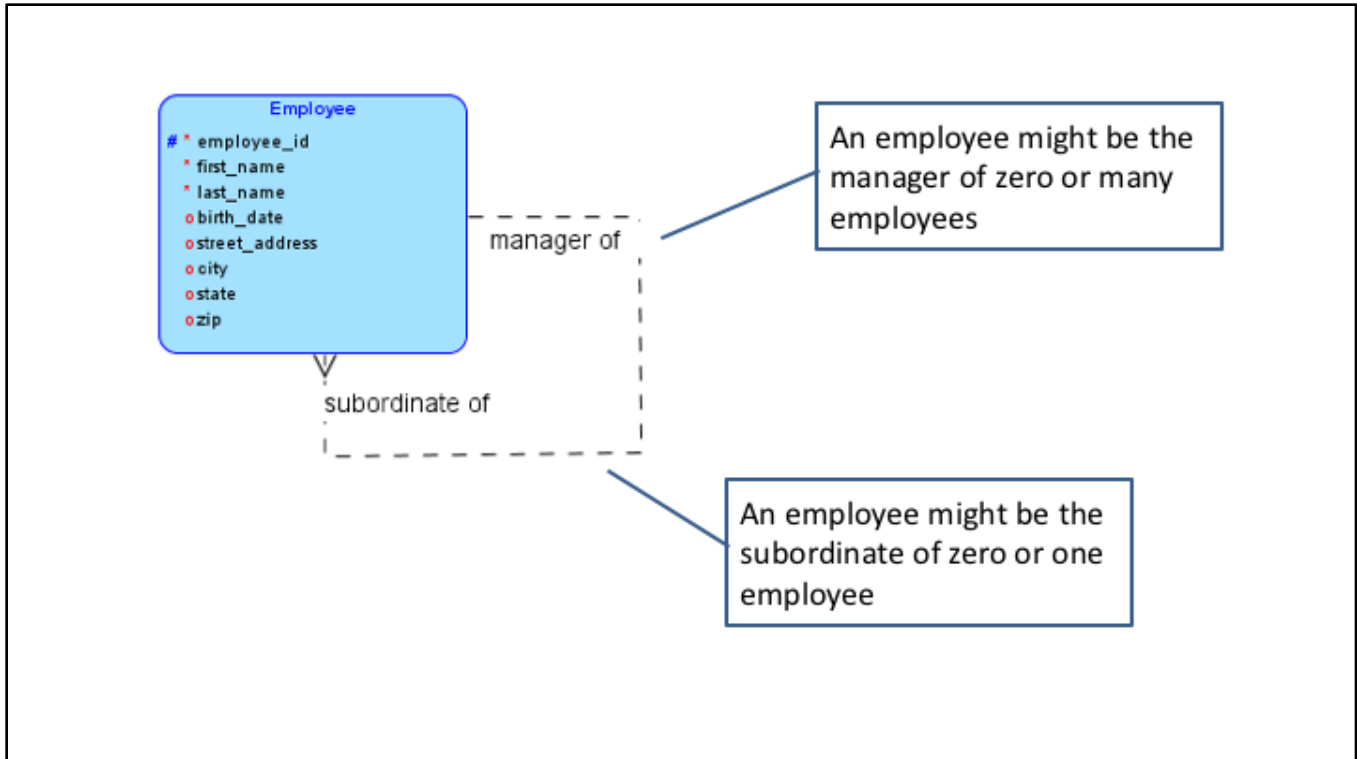
Till now we have seen only binary relationships which connect two different entity types. Although binary relationships predominate, Not all relationships need to be binary.

In a **unary** relationship, instances of an entity type relate to other instances of the same entity type.

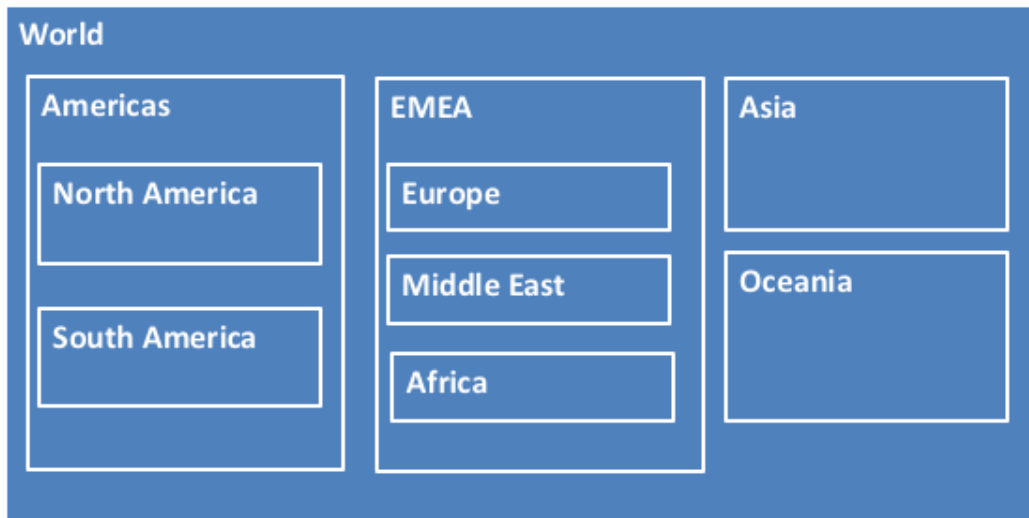
The **UNARY** appears in the name because only one entity type suffices for the relationship.

Note that even though only one entity type suffices, the relationship when seen at instance level connects two different instances of the entity type.

For example, David reports to Melissa – both represent instances of Employee.



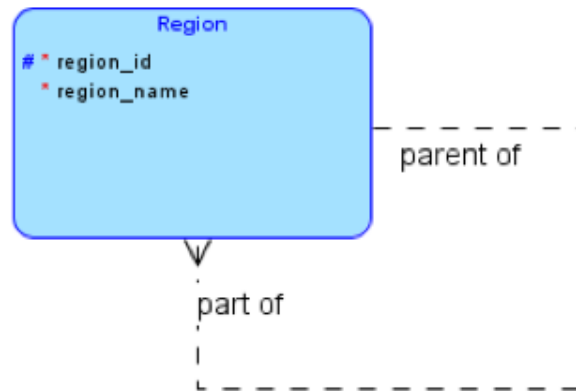
We can read the relationship in two directions using the same approach we saw earlier. To do this we simply traverse the relationship line first in one direction and then in the other.



A multi-national company could view its operations for reporting along geographical lines as shown above. Thus for example, the overall sales could be broken up at the first level as America, EMEA, Asia and Oceania.

However, the Americas region might itself be divided into North America and South America and so on.

Here individual regions are related to other regions. Again we have a **unary** relationship.



A supplier supplies certain quantities of various parts to many projects.

Supplier

Part

Project

Ternary relationship

Note that the statement mentions three entity types. The meaning would not be the same if we divided this into three different parts as below:

Supplier supplies parts

Supplier supplies to projects

Parts are supplied to projects.

None of the above three has the same meaning as the first.

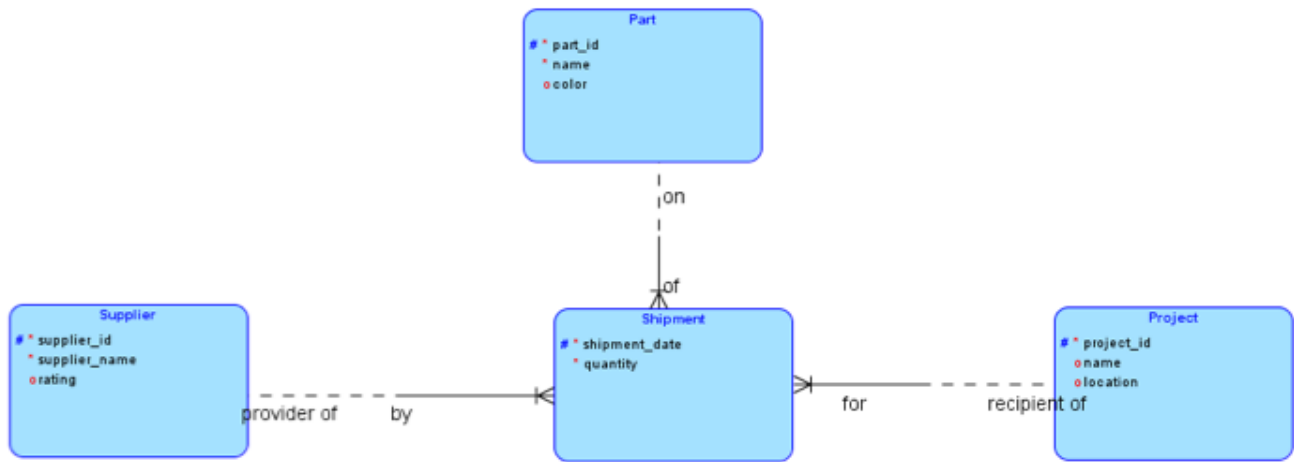
To take a concrete example, suppose we say that supplier A supplied 300 units of part P to project J.

This is not the same as saying that supplier A supplies to project J – which part does he supply?

This is also not the same as saying that supplier A supplied part P – to which project?
And so on.

This relationship connects three entity types – another departure from our cozy world of binary relationships.

Ternary Relationship with Key Migration

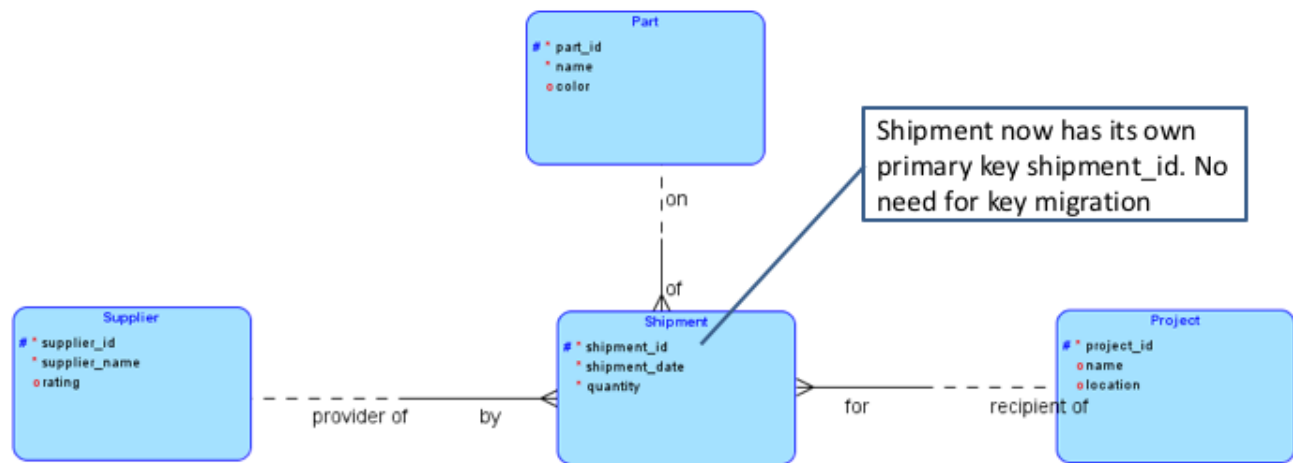


Unlike unary and binary relationships, we cannot show a ternary relationship at all in our notation. Instead, we directly create an associative entity type – Shipment above – and show binary relationships from each of the participating entity types.

In the above example, we have chosen to use key-migration to make up the key for shipment. We also added the shipment date as a key attribute to shipment because it is possible that the same supplier might supply the same part to the same project several times. Thus to allow for this possibility, we have added the date so that the combination of `supplier_id`, `part_id`, `project_id` and `shipment_date` will be unique.

At this point we see that the primary key for shipment has become too unwieldy. In practice when this happens, we do not use key-migration and instead give shipment its own key as shown in the following slide.

Ternary Relationship without Key Migration



Here we have given shipment its own primary key and hence we need not use key migration.

Nevertheless, because of the 1:n relationships, `supplier_id`, `part_id` and `project_id` will still be foreign key attributes in `Shipment`. These are also required attributes because `Shipment` has obligatory participation in all three relationships. We cannot have a shipment without a supplier, part and project.

Shipment therefore has six attributes.

As with earlier associative entity types, every shipment must be associated with a supplier, a part and a project and hence all its lines are solid.

The lines from the entity types `Supplier`, `Part` and `Project` could be solid or dashed, depending on the business rules in force in the situation. Usually we would expect them to be dashed.

A certain public library holds specific items at specific branches for specific patrons.

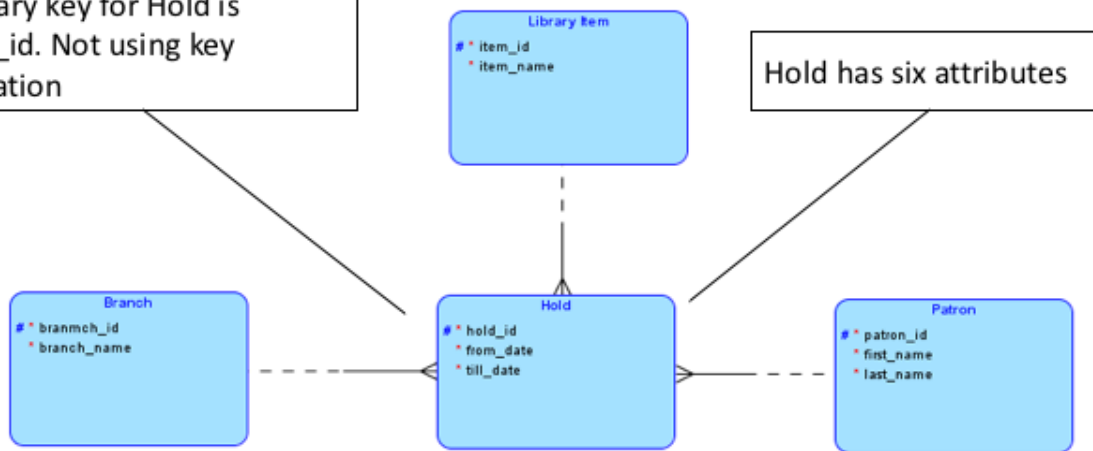
Branch

Library Item

Patron

Hold

Primary key for Hold is
hold_id. Not using key
migration





ERD

Entity type

Visible attributes

Invisible foreign
key attributes

Primary key
column(s)

Database

Table with same name

Columns in table. Required attributes become non-null columns. Others become null-allowed columns

Columns in table. Required or not depending on dashed or solid line on entity type.

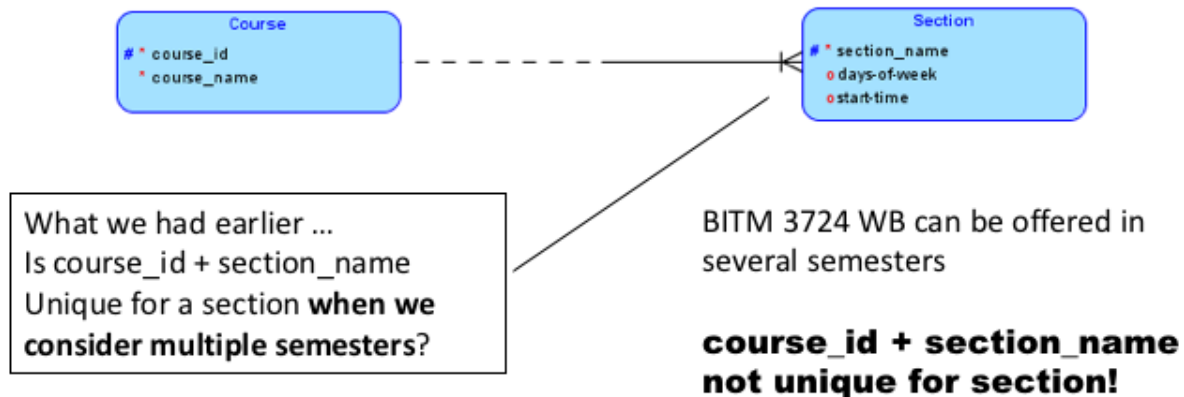
Primary key of table

In a certain university:

- Each course has zero or many sections. Each section is of only one course and a section is identified by a course id and the section name.
- Each section is offered in a particular semester and a semester can have several sections offered in it.
- Each instructor can teach zero or more sections and a section is taught by one or more instructors.
- There are several students and each student can register for zero or more sections and a section can have zero or more students registered in it.

We take this slightly larger example and develop it step by step.

- Each course has zero or many sections. Each section is of only one course and a section is identified by a course id and the section name.



We can see a 1:n relationship with key migration because a section is identified by the course number and section_name ... or is it?

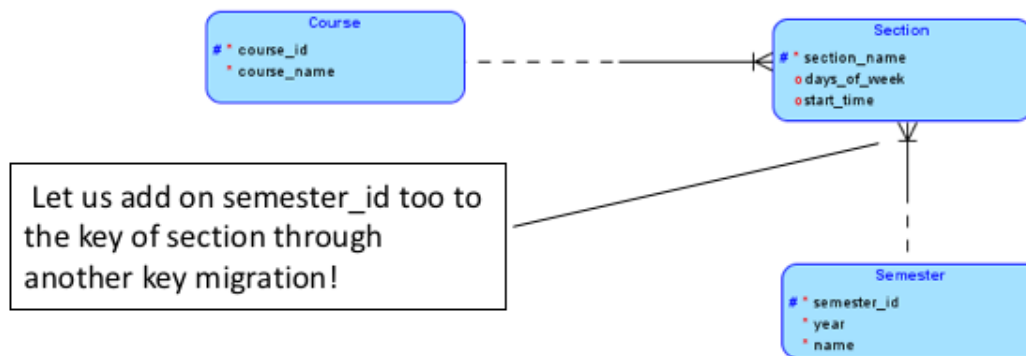
In the earlier cases when we had course_id + section_name as the key for a section, we were not considering a multiple semester scenario. With that in place, the same course_id section_name combination like BACC 4101 AA can be offered in several semesters, for example, in Fall 2012 and in Fall 2013 and in Fall 2014 and so on.

Thus course_id + section_name is insufficient to uniquely identify a section.

What do we do?

Well, the next slide will set this right as it considers the very next bullet point in the requirements.

- Each section is offered in a particular semester and a semester can have several sections offered in it.

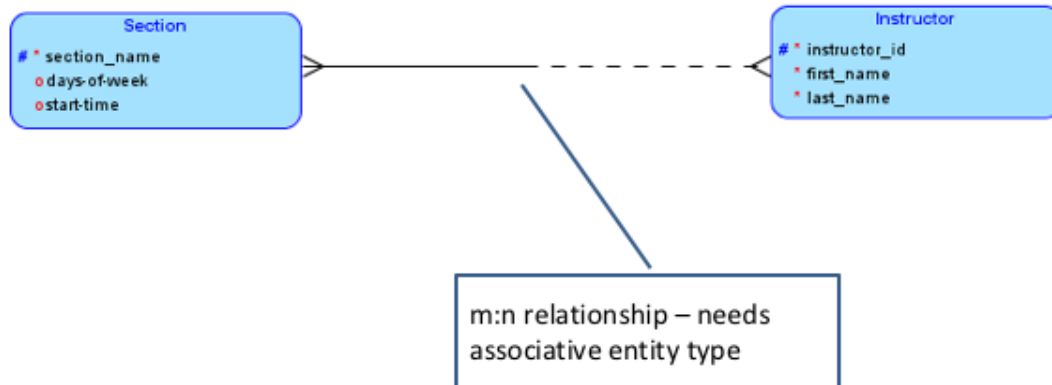


Section is the associative entity between Course and Semester!

We see a 1:n relationship here between Semester and Section. The description does not give a clear indication of obligatory participation and we make reasonable assumptions.

We have added a key migration as well to make the primary key of section to be course_id + section_name + semester_id.

- Each instructor can teach zero or more sections and a section is taught by one or more instructors.

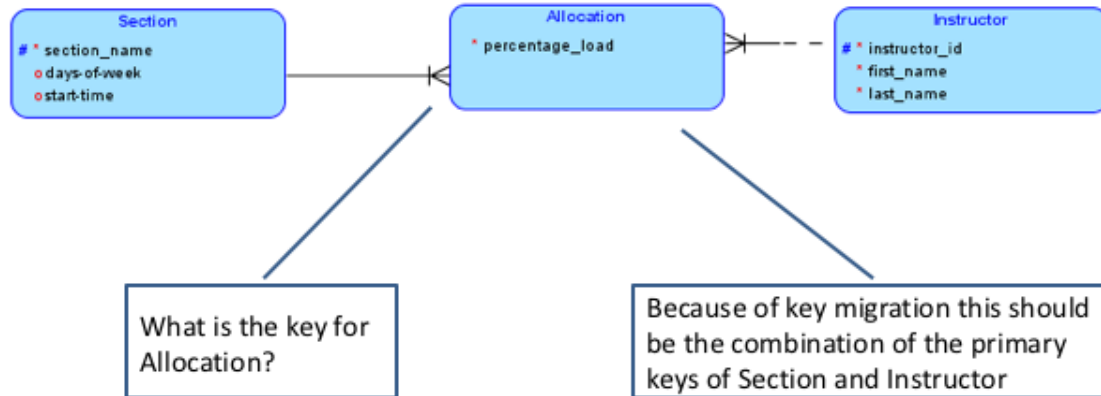


Now we see a m:n relationship between Section and Instructor.

What does this relationship represent in reality? Well, many instructors could be assigned to a single section. That is, a section could be team-taught.

An instance of the relationship represents the allocation of one instructor to one section. We could call the associative entity type as "Allocation". A possible attribute could be "no_of_hours" or "percentage" – let us use the latter.

- Each instructor can teach zero or more sections and a section is taught by one or more instructors.



PK for Allocation: course_id + section_name+semester_id + instructor_id

We have used key migration for the primary key of Allocation.

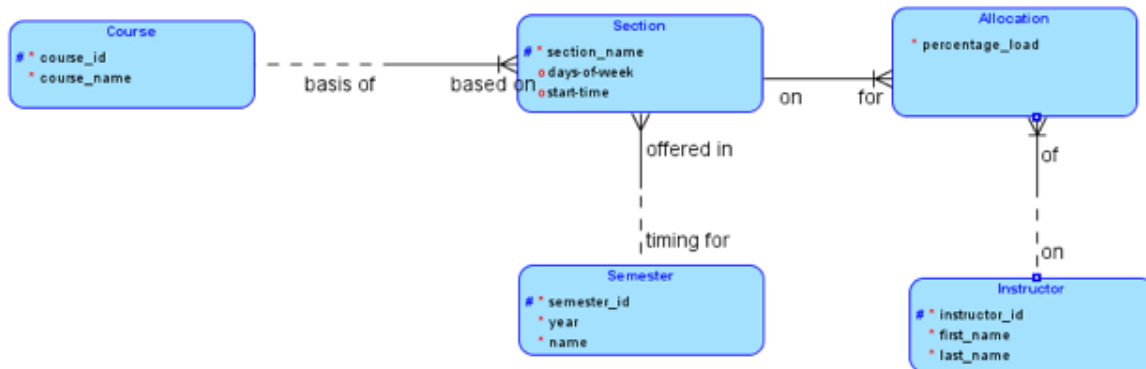
The new thing here is that Section itself is an associative entity type and it is in turn participating in a m:n relationship.

Since we have used key migration, the primary key for Allocation is the combination of the primary keys of Section and Instructor.

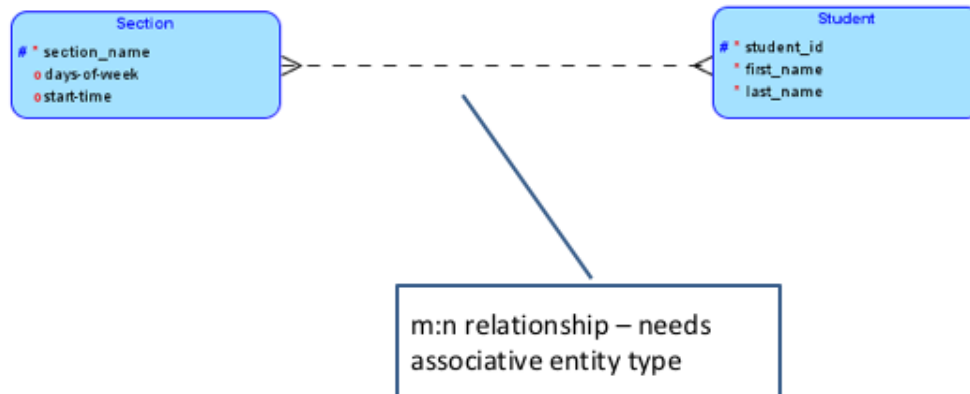
Section in turn has primary key course_id + section_name.

Thus the primary key for Allocation is: **course_id+section_name+instructor_id**

So far we have (with some rearrangement of elements):



- There are several students and each student can register for zero or more sections and a section can have zero or more students registered in it.



Again we see a m:n relationship and need an associative entity type for it.

What does an instance of this relationship represent?

Well, a student is registering for a section. Why not call it Reistration?

registration_date and grade could be attributes.

- There are several students and each student can register for zero or more sections and a section can have zero or more students registered in it.



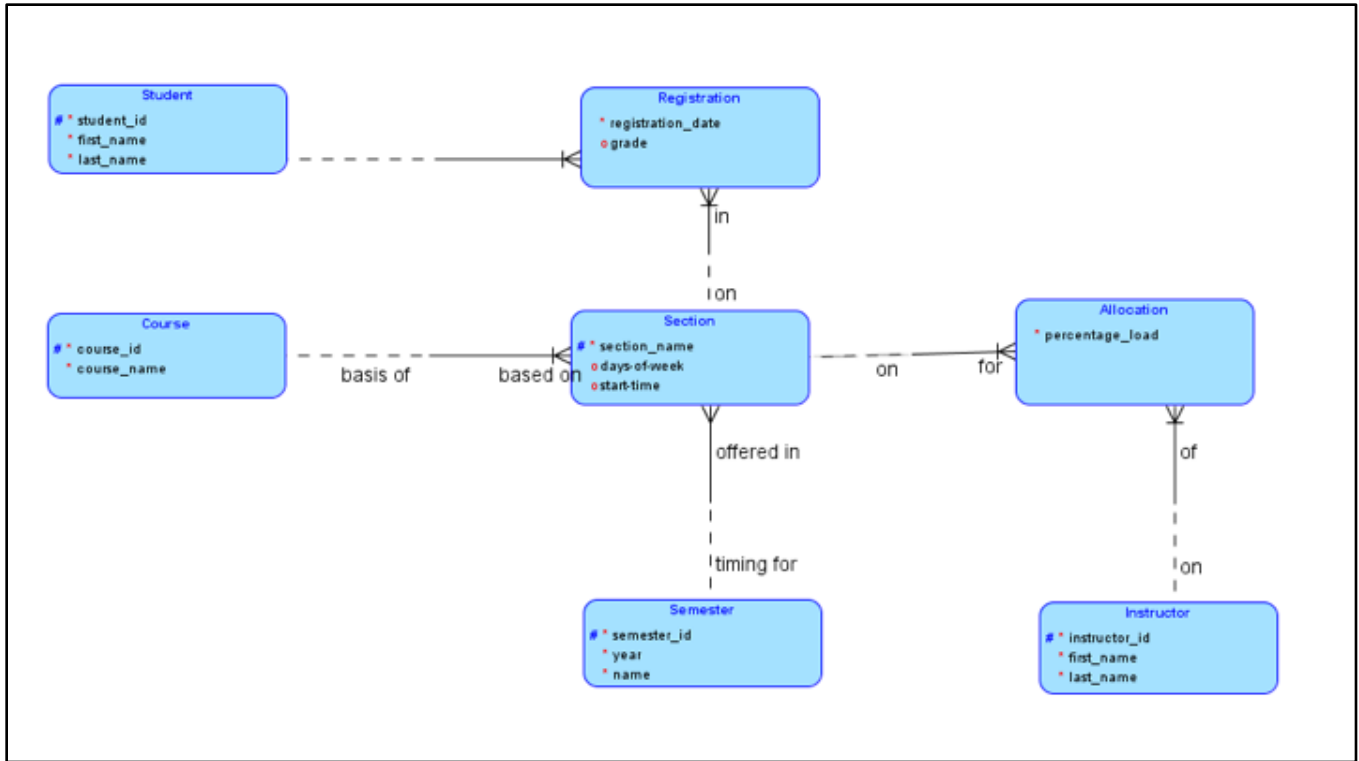
PK for Registration is: course_id+section_name+semester_id+student_id

Again we see a m:n relationship and need an associative entity type for it.

What does an instance of this relationship represent?

Well, a student is registering for a section. Why not call it Reistration?

registration_date and grade could be attributes.



ERD



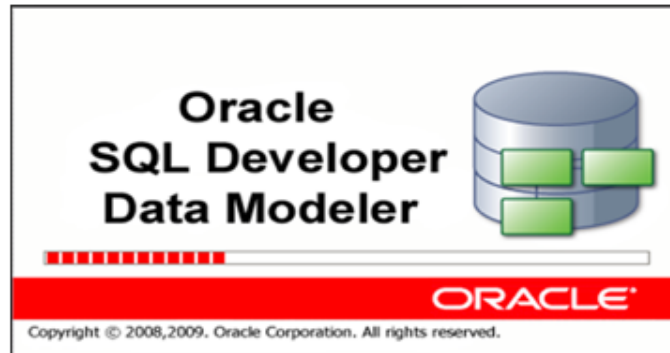
Database

We will now see the code to create the tables for each entity type



We will see some SQL code to generate the tables need to convert our database design into a real, working database.

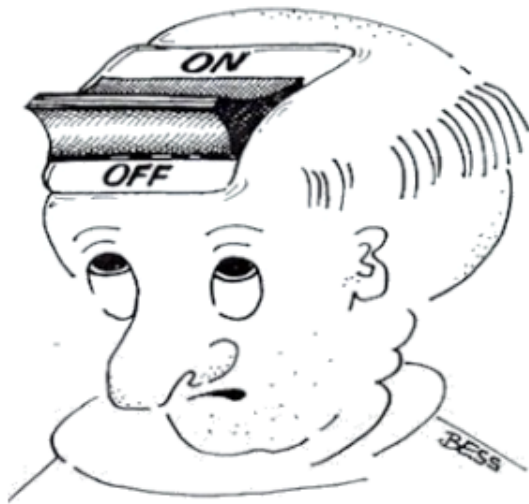
You do not need to memorize any of this ...



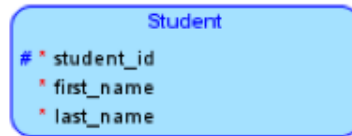
Automates the process

... since we will use Oracle Data Modeler to automate the process.

Nevertheless



... you should not switch off your brains now. You should understand every bit of code that you will see now. Nothin really complex at all, but a few key points that you need to grasp. So pay attention – we will take it one table at a time.



```
CREATE TABLE Student (  
    student_id INTEGER NOT NULL ,  
    first_name VARCHAR2 (50) NOT NULL ,  
    last_name  VARCHAR2 (50) NOT NULL  
);  
ALTER TABLE Student ADD CONSTRAINT Student_PK PRIMARY KEY (  
    student_id  
);
```

The CREATE part specifies the table name and the columns while indicating which columns are required (NOT NULL), and which are optional – no columns in the present case.

The ALTER part then specifies the PRIMARY KEY. The next slide explains.

Primary Key specified as a CONSTRAINT

```
ALTER TABLE Student ADD CONSTRAINT Student_PK PRIMARY KEY (  
    student_id );
```

CONSTRAINTS acceptable values ...

Prevents duplicate values for primary key column(s) in the same table

Primary key is specified as a CONSTRAINT

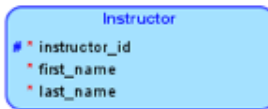


```
CREATE TABLE Course (  
    course_id    INTEGER NOT NULL ,  
    course_name  VARCHAR2 (50) NOT NULL  
);  
ALTER TABLE Course ADD CONSTRAINT  
    Course_PK PRIMARY KEY (  
    course_id  
);
```

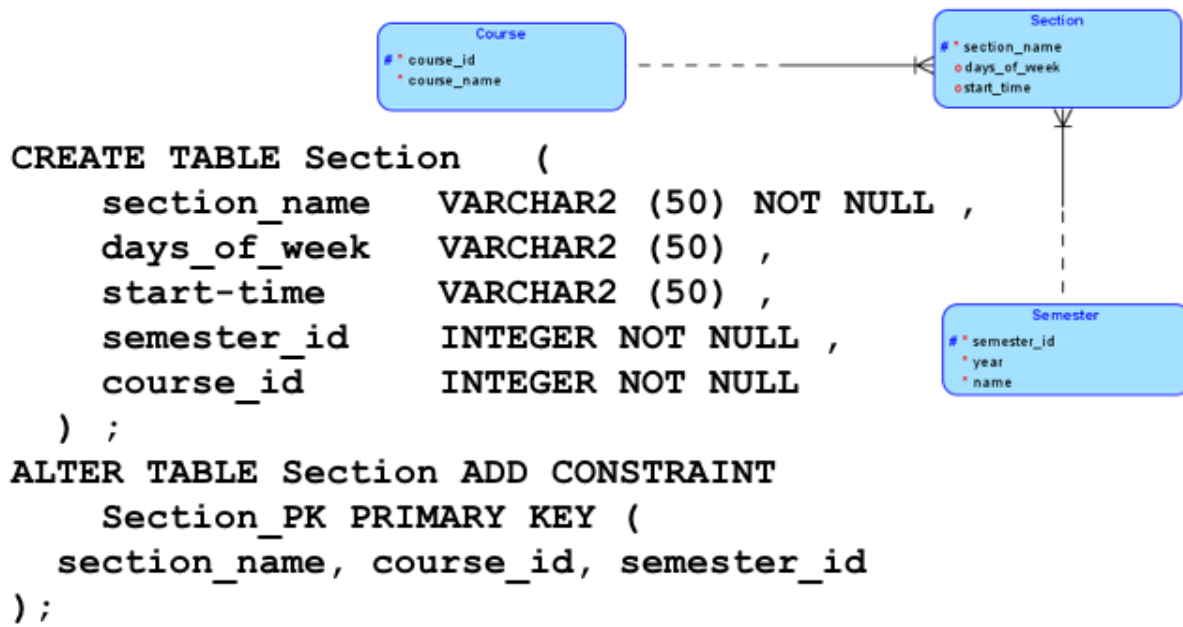

Semester

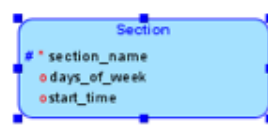
* semester_id
* year
* name

```
CREATE TABLE Semester (
    semester_id INTEGER NOT NULL ,
    YEAR        INTEGER NOT NULL ,
    name        VARCHAR2 (50) NOT NULL
) ;
ALTER TABLE Semester ADD CONSTRAINT
    Semester_PK PRIMARY KEY (
    semester_id
) ;
```



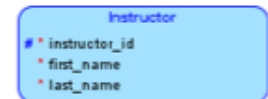
```
CREATE TABLE Instructor (  
    instructor_id INTEGER NOT NULL ,  
    first_name     VARCHAR2 (50) NOT NULL ,  
    last_name      VARCHAR2 (50) NOT NULL  
);  
ALTER TABLE Instructor ADD CONSTRAINT  
    Instructor_PK PRIMARY KEY (  
    instructor_id  
);
```





```

CREATE TABLE Allocation (
    section_name          VARCHAR2 (50) NOT NULL ,
    percentage_load       NUMBER (6,2) NOT NULL ,
    instructor_id         INTEGER NOT NULL ,
    course_id             INTEGER NOT NULL ,
    semester_id           INTEGER NOT NULL
) ;
ALTER TABLE Allocation ADD CONSTRAINT
    Allocation_PK PRIMARY KEY (
        section_name, course_id, semester_id, instructor_id
    );
    
```

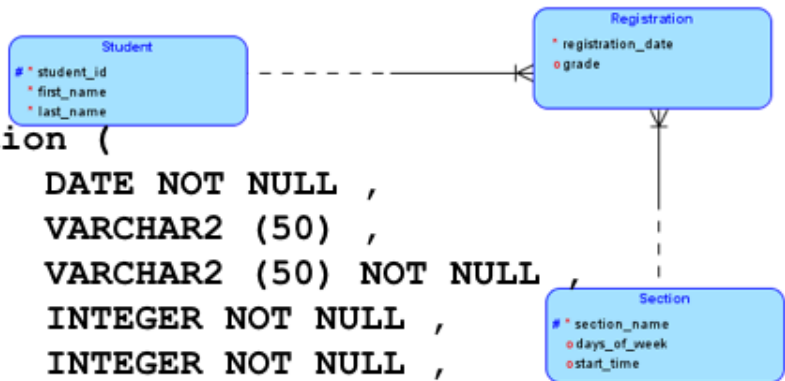


```

CREATE TABLE Registration (
  registration_date  DATE NOT NULL ,
  grade             VARCHAR2 (50) ,
  section_name      VARCHAR2 (50) NOT NULL ,
  student_id        INTEGER NOT NULL ,
  course_id         INTEGER NOT NULL ,
  semester_id       INTEGER NOT NULL
) ;

ALTER TABLE Registration ADD CONSTRAINT
  Registration_PK PRIMARY KEY (
    section_name, course_id, semester_id, student_id
  ) ;

```

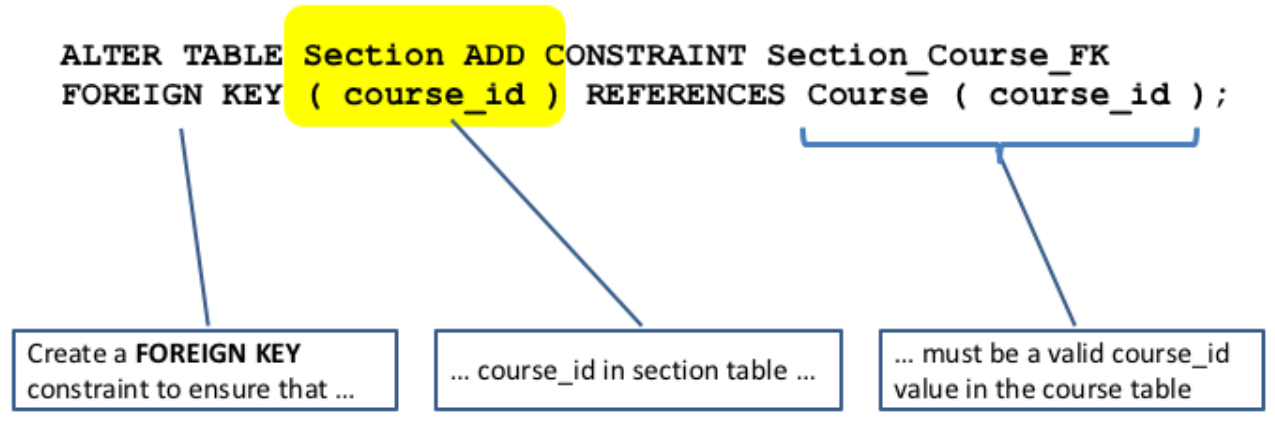


Foreign Key specified as a CONSTRAINT

Every course_id in section table must be one of the values in the course table

Cannot have section of a non-existent course!

```
ALTER TABLE Section ADD CONSTRAINT Section_Course_FK  
FOREIGN KEY ( course_id ) REFERENCES Course ( course_id );
```



Create a **FOREIGN KEY** constraint to ensure that ...

... course_id in section table ...

... must be a valid course_id value in the course table

Here is the complete list of FOREIGN KEY constraints:

```
ALTER TABLE Allocation ADD CONSTRAINT Allocation_Instructor_FK FOREIGN KEY (  
instructor_id ) REFERENCES Instructor ( instructor_id );
```

```
ALTER TABLE Allocation ADD CONSTRAINT Allocation_Section_FK FOREIGN KEY (  
section_name, course_id, Section_semester_id, semester_id ) REFERENCES Section (  
section_name, course_id, semester_id );
```

```
ALTER TABLE Registration ADD CONSTRAINT Registration_Section_FK FOREIGN KEY (  
section_name, course_id, Section_semester_id, semester_id ) REFERENCES Section (  
section_name, course_id, semester_id );
```

```
ALTER TABLE Registration ADD CONSTRAINT Registration_Student_FK FOREIGN KEY (  
student_id ) REFERENCES Student ( student_id );
```

```
ALTER TABLE Section ADD CONSTRAINT Section_Course_FK FOREIGN KEY ( course_id )  
REFERENCES Course ( course_id );
```

```
ALTER TABLE Section ADD CONSTRAINT Section_Semester_FK FOREIGN KEY ( semester_id )  
REFERENCES Semester ( semester_id );
```

Create CONSTRAINTS

Exploit automatic enforcement

Maintain database INTEGRITY

CONSTRAINTS maintain database INTEGRITY

Integrity Constraints

Integrity Constraints

Primary key
constraints



Entity integrity

Foreign key
constraints



Referential integrity

```
ALTER TABLE Allocation ADD CONSTRAINT  
Allocation_Instructor_FK FOREIGN KEY (  
instructor_id ) REFERENCES Instructor (  
instructor_id ) ;
```

```
ALTER TABLE Allocation ADD CONSTRAINT  
Allocation_Section_FK FOREIGN KEY ( section_name,  
course_id, semester_id ) REFERENCES Section (   
section_name, course_id, semester_id ) ;
```

```
ALTER TABLE Registration ADD CONSTRAINT  
Registration_Section_FK FOREIGN KEY ( section_name,  
course_id, semester_id ) REFERENCES Section (  
section_name, course_id, semester_id ) ;
```

```
ALTER TABLE Registration ADD CONSTRAINT  
Registration_Student_FK FOREIGN KEY ( student_id )  
REFERENCES Student ( student_id ) ;
```

```
ALTER TABLE Section ADD CONSTRAINT  
Section_Course_FK FOREIGN KEY ( course_id )  
REFERENCES Course ( course_id ) ;
```

```
ALTER TABLE Section ADD CONSTRAINT  
Section_Semester_FK FOREIGN KEY ( semester_id )  
REFERENCES Semester ( semester_id ) ;
```


Database Design



This completes our discussion of database design.

Complete the hands-on exercise

“Step-by-step process for using Oracle Data Modeler for drawing ERDs and generating database scripts” before you start on the assignment for this week.