# **CS579 Fall 2009**

# Class #4 Homework

# Hardcopy only (no e-mail) due at start of Class #5

**Submitter: Charlie Kim**

Reading

**Bowman (The Practical SQL, 4th Ed)**

* Chapter 7 (Joins), pages 205-233

**Models and Languages for Database Design & Theory**

Visual & Textual Languages for ER Modeling

1.6-1.8 Attribute Domains & Constraints

1.11 Derived Attributes

3.1-3.3 Dependent & Identifying Relationships

**Connolly & Begg**

* Section 11.4 ER Modeling (Strong & Weak Entity Types)
* Section 5.3.7 SQL (Multi-Table Queries)
* Section 12.2-12.3 Enhanced ER Modeling (Aggregation & Composition)

**Oracle Database 10g**

* Chapter 13 Query Dependence, pp 249-253 (Joins)

Lab Exercise #4 (complete before class #5)

Available online at courseinfo.

Do NOT hand in the output.

Project Continuation

🡺 Note: **Your team name and/or project name must appear on all project documents**. You can change your team or project name if you need to (but let me know)

🡺 Look at my Example Project Deliverable for Week #4, and please use the same format.

This week, your team must hand in (stapled together **separately** from any homework problems):

**1)** (if your team has > 1 person) A team participation report (written by the team leader), listing the team name for your group, the name of your project (if different from the team name), the name of the team leader, and for each member of the team

* a very brief description of the main things that member focused on this week
* the percentage of the overall work that member did this week

**2)** A revised set of project requirements. **Highlight any changes.**

**3)** A revised ER diagram, using either (Easy) Crow Magnum or UML, for the entity classes needed by your project. The ER diagram must

* Include 5-9 entity classes
* Include at least as many 1:M and M:N relationships as there are entity classes (count carefully!)
* 🡺 **Include at least one weak entity class, which has at most one identifying relationship**
* Show **ALL** attributes (with primary keys and discriminators properly denoted) and **all** mandatory participation constraints (do NOT show optional participation)

**4)** A list of the entity classes, with a very brief description of each.

**5)** A revised user role diagram. **Highlight any changes.**

**6)** A list of User Operations for your application.

* Organize them by role.
* Within each role, separate the queries from the actions.
* For each operation, include the parameters and a very brief description. Use a format similar to the ones used in previous assignments.
* Do not include more than 25 operations, so if you can think of more, pick the 25 most important and useful ones. You will probably only be implementing 10-15 of them, but you don't need to choose those yet.

In addition

* You should have both a Login and a Logout operation. Login should take both a userid and a password as parameters. The userid could either be numeric or a string, depending on what you think is most appropriate or convenient for your application.
* You must have at least one action operations (other than Login, Logout or SetPwd) and two query operations whose functionality depends in some way on the **identity** (not just the role) of the user who is currently logged in (but not just to enforce some pre-condition). For example, in the Employee Database Application ChangeAddr changes the address of the *current logged in user*, and ShowDeptEmps shows the names and jobs of all employees who work in the same department as the *current logged in user.*
* You should have at least one action operation, and at least one query operation which requires access to data in two or more entity classes.
* Your application code will keep track internally of the identity of the user who is currently logged in. The means that 🡺 you should almost never provide the current user as a parameter to an operation (other than Login), since the operation's implementation can find out (on its own) who is the current user.
* Do not include operations which require interaction with the user (e.g. via a dialogue box) *after* the user has entered parameters and invoked the operation). If you have an operation which you believe requires such interaction, 🡺 you need to discuss how to handle this situation with me.

**7)** List at least **one** interesting relationship or general conceptual state constraints (with **names** for each) that could be enforced by your application. Each constraint

* must be described in plain non-technical English, and refer to entities and relationships in the conceptual ER model (not to tables)
* involve at least one relationship and/or two entity classes
* must NOT be a constraint which could be represented directly in the ER diagram itself (e.g. you MAY NOT use a dept must have at least 3 employees, since that can be represented directly in the ER diagram)
* must not be able to be implemented just by using a built-in relational constraint (e.g. UNIQUE, CHECK, NOT NULL, REFERENCE).
* could potentially be violated by one of your operations, depending on the parameters to the operation, the identity of the user, the current state of the database (even if it doesn't initially violate the constraint), and/or the order in which operations are executed.
* must correspond to or be implied by some statement in your project requirements (add it if necessary)
* MUST NOT BE identical to one in the Employee Database Application

🡺 For each of these constraints, list

* the user action operations which could potentially violate the constraint.

**8) [EXTRA CREDIT]** Identify at least **two** actions which have *interesting* **state-related post-conditions**. Remember that post-conditions are interesting if they have some of the following characteristics

* Don't obviously or immediately follow from a simple description of the operation
* are not invariant (i.e. always true); the operation made it true
* involve changes to entity classes or relationships other than the one that the operation obviously needs to affect
* ensure that redundant data is kept consistent

List the two actions and the post-conditions of those actions. Make sure that all of the post-conditions are implied by some statement in your project requirements (add it if necessary)

* At least one of the post-conditions should affect a class or relationship other than the one(s) primarily affected by the operation.
* 🡺 Include at least one post-condition which is not simply a lifetime dependency rule (e.g. when a dept is deleted, delete all of its employees). Often, this can be done, by having the side-effect either  
  + depend on a change of state, rather than on deletion -- so for example, in the Employee Database Application, we could add a status attribute to the Department entity class. A post condition for the operation that sets the status of a department to INACTIVE could require that employees of the department are deleted.
  + associate the child with another parent rather than deleting it -- so for example, in the Employee Database Application, when a department is deleted, the post condition could move its employees to another department (e.g. the PARTY dept, or the department with the smallest # of employees)
  + be based on the deletion of the last child (of some type) of a parent -- so for example, in the Employee Database Application, when the last clerk in a department is deleted, the post condition could move a clerk to that department from another dept which has two of them, or else it might change some other employee's job to clerk.

**9) [EXTRA CREDIT]** Identify **at least** **one** operation (preferably two, and not Login, Logout or SetPwd) which have **identity-based** **user pre-conditions**. List the actions and the pre-conditions.

A **user pre-condition** describes a constraint on the circumstances in which an operation is allowed to execute, based on the role or identity of the current user. User pre-conditions may either be role-based or identity-based.

* A **role-based** user pre-condition depends just upon the *role* of the user who is logged in -- for example, in the Employee Database Application, ShowDirect is associated with the Manager role. This is equivalent to the pre-condition for ShowDirect: the current user must be a Manager (or some subrole of Manager). 🡺 Don't include role-based pre-conditions for this deliverable.
* An **identity-based** userpre-condition depends upon the identity of the user who is logged in, in some way more complicated than just their role. For example, in the Employee Database Application, TerminateEmp, which terminates an employee, can only be done by the President, or by the Dept Manager who is in the *same department as the* *current logged in user* (i.e. not just by *any* user with the role of DeptMgr)

**PROJECT GRADING**

Each individual's grade for the project will be a combination of (a) the grade for the project as a whole, and (b) the contribution that you made to the project (in terms of quality, quantity, and leadership). Individual project grades will be determined at the end of the term.

It is your individual responsibility to ensure that you have an opportunity to participate in and review each aspect of the project, since that will both affect your project grade, as well as your ability to do well on exams. If there are any problems or issues, let me know as soon as possible.

It is the team leader's responsibility to make sure that (a) the team succeeds, and (b) you do your best to see that each person on the team has the opportunity to participate in and/or review each aspect of the project. If you have any concerns about your team's success, or the success of any member of your team, it is your responsibility to let me know as soon as possible. Either the team or I can change the team leader if necessary.

Self Quiz

Do not hand these in. Make sure you know the answers, though!

a) Explain the similarities and differences among the results of the following queries

SELECT ename, pname FROM Emps, Projects  
 WHERE empno = pmgr;

SELECT **e.**ename, **p.**pname FROM Emps **e**, Projects **p**  
 WHERE **e.**empno = **p.**pmgr;

SELECT **e.**ename, **p.**pname  
 FROM (Emps e CROSS JOIN Projects p)  
 WHERE **e.**empno = **p.**pmgr;

SELECT ename, pname  
 FROM (Emps **e** CROSS JOIN Projects **p**)  
 WHERE empno = pmgr;

b) Explain the similarities and differences among the results of the following queries

SELECT ename, pname  
 FROM (Emps CROSS JOIN Projects)  
 WHERE empno = pmgr  
 AND sal > 2500;

SELECT ename, pname  
 FROM (Emps JOIN Projects ON empno = pmgr)  
 WHERE sal > 2500;

SELECT ename, pname  
 FROM (Emps JOIN Projects   
 ON empno = pmgr  
 AND sal > 2500);

Problems

IMPORTANT: You may do any of these problems on your own or with a partner.

About working with a partner or team:

* If you do a problem together with a partner or team, then only hand in ONE COPY of that problem with all of your names on it.
* Please staple together (no paper clips please, they detach way too easily) the problems you did alone. Separately staple together the problems you did together.
* Working with a partner or a team means working *together* on the entire problem.  If you just start working together, but then actually work out the details on your own, hand in your own solution.  If you personally did very little work on a problem, but your name is on it when it is handed in, then you are involved in plagiarism.
* If you are working with partners, and there's a significant imbalance in the quantity or quality of a partner's work, it’s essential you talk to me. Otherwise, it will almost certainly be the case that the underperforming partner will end up doing badly in the exams, and will either be forced to drop the course or will end up with a low grade. If you let me know there's a problem, I can make sure that students get additional help if they need it, or work with a different partner or team.

If you have ANY questions, ***post them on the newsgroup***. Answering questions posted to the newsgroup can positively impact your grade.

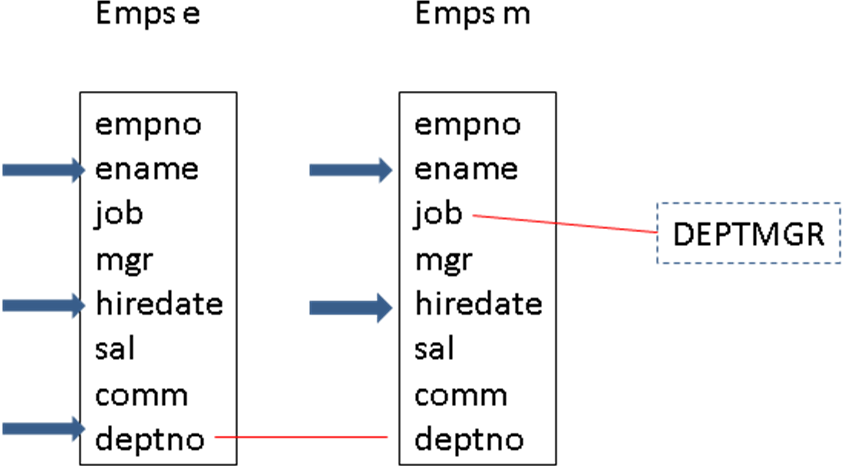
**1. [30%] SQL Join Problems**

Solve each of the following problems that involve the Emps relation. This exercise is primarily about using joins, so for those of you who know about subqueries, DON'T USE THEM.

For each problem, you MUST write a single SQL SELECT statement which produces the requested result (I recommend you test it, but that's not required)

**a)** Find the employees in each department who were hired after their department manager (i.e. the one person in the employee's department with a job of 'DEPTMGR'; this need not be their direct manager!). For each such employee, list their name, department number, hiredate, and the name and hiredate of the department manager. Sort the result by department number, and within that by the employee's hiredate. There are two parts to this problem:

**a1)** Draw the join diagram



**a2)** Write the SQL SELECT statement. For full credit, use JOIN … ON to identify the matching tuples

**SELECT e.ename, e.deptno, e.hiredate,**

**m.ename as mname, m.hiredate as mhiredate**

**FROM ( Emps e JOIN Emps m**

**ON e.deptno = m.deptno AND m.job = ‘DEPTMGR’ AND e.hiredate > m.hiredate )**

**ORDER BY e.deptno, e.hiredate ;**

**b)** Write the SQL to list ONLY the employee numbers (NOT the employee names) of those who directly manage more than one other employee (think about this carefully!)

**SELECT mgr FROM Emps GROUP BY mgr HAVING count(\*) > 1 ;**

**c)** Write the SQL to list the names (but NOT employee numbers) and jobs of employees who directly manage other employees, along with the count of the employees they directly manage. Extra credit if you consider the possibility that more than one employee may have the same name and job.

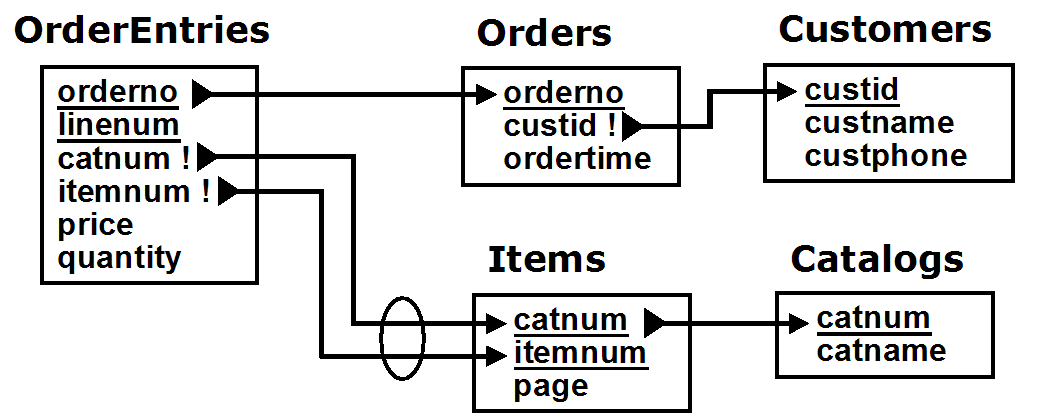
**SELECT m.empno, min(m.ename) as ManagerName,**

**min(m.job) as ManagerJob, count(\*) as numemp**

**FROM ( Emps e JOIN Emps m ON e.mgr = m.empno )**

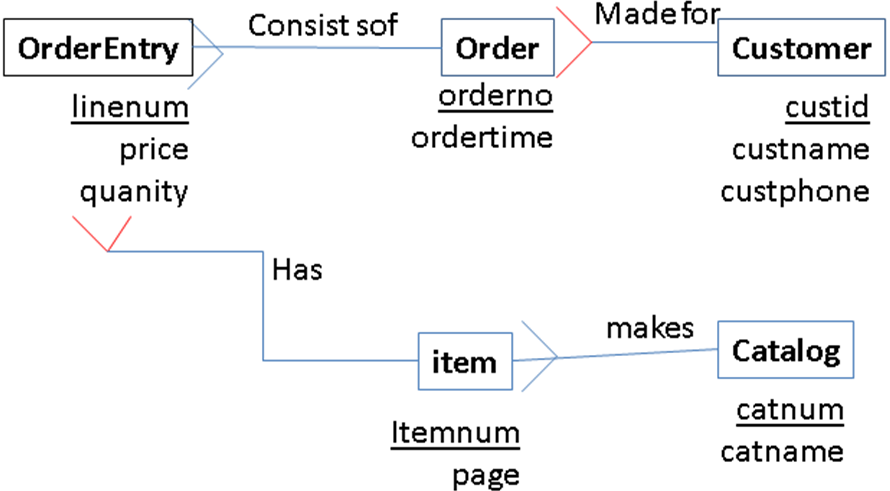
**GROUP BY m.empno ;**

**2. [20%] Weak Entity Class Reverse Engineering**



Look at the Relational Schema above for a system which maintains unprocessed customer orders.

Draw an Easy Crow Magnum or a UML ER Diagram (showing all attributes) that would naturally map to it. Make sure you use the appropriate symbol to indicate in the ER diagram which relationships should be independent, mandatory, dependent, or identifying, based on the relational state constraints in the diagram above.



**3. [40%] Conceptual Design Exercise**

There are 3 parts to the following problem: **(a)**, **(b)**, and **(c)**. Consider the requirements below, and

**a)** Draw an Easy Crow Magnum or UML Diagram that meets the requirements, described below, showing

* all relationship characterizations
* all attributes (but only those reasonably implied by the requirements) and
* all necessary mandatory (but not optional) participation constraints.

Some important considerations

* Make sure you use dependent and identifying relationships where appropriate. You will gain credit by doing so, but only if done correctly.
* UML & Crow Magnum indicate derived attributes by preceding them with a slash. You will gain credit by indicating derived attributes correctly, but only where appropriate.
* Review the Fan Trap and the Reverse Fan Trap. You can easily fall into one the fan traps in doing this problem. Avoid that!

**🡺 b)** Specify all additional conceptual state constraints involving dates (but not required constraints – don't include constraints that say a date attribute is required)

**🡺 c)** Specify additional ***conceptual*** state constraints required (or clearly implied) by the requirements (i.e. **don't just make stuff up!**), and that can be supported by the entities, attributes and relationships in the ER diagram

* List all other (non date-related) *conceptual* state constraints specified in the requirements below that are not already represented in the ER diagram, however
* Do NOT include *relational* state constraints or assertions (written using SQL)
* Do NOT include business rules which are not state constraints (e.g. pre- and post-conditions).
* Do NOT include state constraints that are implied by your ER diagram.
* Do NOT include {required} constraints (i.e. requiring that some attribute's value must be provided -- i.e. can't be null)
* Do NOT include {unique} constraints (i.e. requiring that all the values of some attribute be unique)

The requirements are

* A video store wants to keep track of all of its videos. When it acquires a copy of a video, it stamps a *unique* vidid number on it, and uses the vidid to keep track of the video.
* For each individual video it owns, the store wants to be able to track the date the video was acquired, the name of the distributor of the video, the title of the video, the year in which it was made, the id, name, address and phone number of the customer (**if any**) who ***currently*** has it rented along with the date and the time the video was rented, its due date, and the employee number and name of the employee who processed the rental. Keep track of current rentals only; there's no need to keep track of a video's previous rentals.
* Every customer has a membership category; the category determines the number of days that customer is allowed to rent videos.

Your design MUST use *at least* the following entity classes: Film (a film title), Video (a particular copy of a film title), Distributor, Employee, and Customer. By the way, note that because of remakes, two films could have the same title, but would be made in different years.

Also, your design MUST use *at least* the following attributes: vidid, distributor, title (of the film), yrmade (the year the film was made), custid, custaddr, custphone, category (the category of a customer), acqdate (theh date a video was acquired from the distributor), rentdate (the date a video is rented, renttime (the time of day a video is rented), duedate, empno, ename.

Note:

* + You may add other entity classes. It is not required, but do it correctly and you will discover that
* the overall design will be better
* additional conceptual constraints can be represented directly in the ER diagram, so you don't have to list them explicitly.  
  + You may add other attributes, but ***no added attributes may be primary keys or discriminators.*** You will LOSE CREDIT if added attributes are primary keys or discriminators.

Finally, here are some additional requirements. Each of these requirements MUST either be represented directly in the ER diagram itself (for more credit) or by specifying a conceptual state constraint.

* All individual videos are obtained from distributors (i.e. the store doesn’t make its own copies), and all copies for a particular film are distributed by a single distributor only.
* A customer can rent a group of videos at a single time (with the same rentdate and renttime); these are all processed by the same employee.

**4. [10%] Relational State Constraints**

a) Write the table CHECK constraint corresponding to the conceptual constraint below. Note: this is the CHECK constraint that you would write as part of the definition of the Emps table, NOT an arbitrary assertion.

Every dept mgr is assigned to a department

**CHECK( job != ‘DEPTMGR’ OR deptno IS NOT NULL )**

**b) [EXTRA CREDIT]** Write the relational state assertion (NOT the check constraint) corresponding to the conceptual state constraint [Note: the answer is different from (a)]. You can assume that all employees have a job.

Every dept mgr is assigned to a department

**CREATE ASSERTION NoNodeptDeptmgr**

**CHECK(**

**(SELECT deptno FROM Emps WHERE job = ‘DEPTMGR’) ALL IS NOT NULL**

**)**

**c) [HARD EXTRA CREDIT]** Write the relational state assertion (definitely MUCH harder than you might think!) corresponding to the conceptual state constraint:

Every department has exactly one department manager

(You can assume the every dept mgr is assigned to a dept, but you CANNOT assume that every dept has at least one dept mgr)

**CREATE ASSERTION OnlyOneDeptmgrPerDept**

**CHECK(**

**( SELECT count( distinct deptno ) FROM Emps ) =**

**( SELECT count( distinct deptno ) FROM Emps WHERE job = ‘DEPTMGR’ )**

**)**