CS145 Final Exam Solutions

Index

Problems 1: (b) and 2: (c)

First, what are the keys? *C* must be in every key, so it is not hard to deduce that the only keys are *ABC* and *CD*. Since *E* is therefore the only nonprime attribute, none of the given FD's violate 3NF. However, *D->A* and *D->B* each violate BCNF, so the answer to 1 is (b).

The computation of the number of superkeys is exactly the same as a problem on HW2; it uses the principle of inclusion/exclusion. That is, there are 4 superkeys containing *ABC*, 8 superkeys containing *CD*, and 2 superkeys containing all of *ABCD*. Thus the number of superkeys is 4+8-2 = 10.

Problem 3: (a)

This problem is quite similar to one on the midterm. Remember that the meaning of an arrow in E/R diagrams is ``all other entity sets determine this one.'' Thus, the only violations are two tuples of the relationship set that agree in A, B, and C but disagree in D, or that agree in A, B, and D but disagree in C. Only choice I has such a violation, so the answer is (a). If you believed that it was sufficient to have two tuples that agree in A and B but disagree in either C or D, then you chose (c), which is wrong.

Problem 4: (d)

B->->A follows from the complementation rule, while *A->->C* follows from first promoting the FD to an MVD and then using the transitive rule for MVD's. Thus, II and III hold. To see that I (*B->C*) doesn't hold, consider the relation

A	B	C
a	b	c
a	b	d

which satisfies the given dependencies but not I.

Problems 5: (a) and 6: (b)

``ODL-style'' wins in situations where we want to ask about a subclass and attributes that belong to that subclass and also to some superclass. Query (a), which talks about the drive (and therefore can pertain only to SUV's) and also talks about engines (which are found in all automobiles), is the only one to meet this condition, and thus is the answer to Problem 5.

On the other hand, ``E/R-style'' wins when we have a query about attributes that can belong to two or more classes, for ODL-style will then require us to look in several relations, while E/R-style lets us look only in the superclass. Choice (b) is the only one that meets this description, so it is the answer to Problem 6.

Note that choice (c) is easy in either database schema.

Problem 7: (d)

Expressions (a), (b), and (c) each compute what is called the *semijoin* of R with S, that is, the set of tuples of R that join with one or more tuples of S. Equivalently, the semijoin is the nondangling tuples of R in the join R JOIN S. However, (d) produces all of R, regardless of S (except in the case when S is empty; see Problem 8). Thus, (d) is the answer.

Problem 8: (c)

Equation (a) is false in the case that S is empty. In that case, the left side is empty, but the right side is R.

Equation (b) gets the naming of attributes right, but the left side computes R-S, while the right side computes S-R.

However, (c) is a true statement; it expresses the relationship between the natural join and a similar equijoin.

Problem 9: (b)

Not much to say here. You just had to remember that in a many-one relationship, the ``many'' side has a relationship whose type is a collection, while the ``one'' side does not.

Problem 10: (c)

First, II is a direct expression of the problem, and is correct. III misses the condition ``no common parent'' and thus is not correct. I is not correct either, but it is harder to see why.

Abstractly, I is saying that x and y are cousins if they have *some* parents that are not the same. More concretely, here's a family history that is bizarre but legal in most states. Sally and Sue are sisters; let Gus be their common father. Joe marries Sally and has child Alice. Joe later divorces Sally and marries Sue (and opens a bar --- but that's another story). Joe and Sue have child Bob. Now Alice and Bob are siblings; they have common parent, Joe. Thus, they are *not* cousins according to the problem statement (or in real life). However, rule I lets us deduce `Cousin(Alice,Bob)` if we let x = Alice, y = Bob, xp = Sally, yp = Sue, and z = Gus.

Problem 11: (c)

II is a straightforward definition of the conditions, but I fails because a foreign key must reference a primary key, not just something that is unique.

Problem 12: (a)

Not much to say. Both queries are common SQL ways to express the requested idea.

Problem 13: (d)

I is obviously wrong, since the subquery is not connected to the outer query in any way. You get all manager ID's provided there is at least one employee at the company whose name isn't Sally.

The problem with II is more subtle; it is a confusion of ``there exists'' with ``for all'' akin to the trouble with the Datalog rule in Problem 10(I). Suppose Sue manages Sally and Joe. We do not want to produce Sue's ID among the answers, because she *does* manage a Sally. However, when the `Emps` tuple in the outer query, say t, is the tuple for Joe, then the where-condition of the outer query is true. That is, `EmpID`, being the employee ID of Joe, will not be equal to any Id of an employee named Sally. Thus, Sue's ID, the one in the `mgrID` field of tuple t, gets printed.

Problem 14: (a)

Both assertions are straightforward ways to express the desired constraint. I says that when you group employees by their manager, you do not have any groups of more than 10 tuples. II says that if you construct a view consisting of managers and the counts of their employees, you do not get any counts greater than 10.

Problem 15: (b)

I is a straightforward implementation of the query. II differs from I in two ways. One way is that the chains are grouped in the opposite way, but it doesn't matter. That is, whether you express a path recursively as an arc followed by a path (as in I) or as a path followed by an arc (as in II) is irrelevant. The second change is in the basis case, and here II goes astray. Since has four attributes, it cannot match the second term of the union, which has only two.

Problem 16: (d)

The trick to finding counterexamples for bag laws involving union, intersection, and difference is to think of bags with only one element x, appearing multiple times. Then the bag becomes a number (the number of times x appears), union is addition, intersection is min, and difference is proper subtraction (subtraction but not below 0). Then (a) becomes i+i = i, which is false for any i>0. Choice (b) is also false. For example, let R, S, and T each have 1 x. Then the left side is 1 and the right side is 2. Finally, (c) is also false. For instance, let R, S, and T have 3, 2, and 4 x's, respectively. Then the left side is 3+(2-.4) = 3+0 = 3, while the right side is (3+2)-.4 = 5-.4 = 1.

Problem 17: (d)

The two tuples (1,2) combine with the two tuples (2,5) to produce four tuples (1,2,5). The tuple (3,4) combines with (4,6) to produce (3,4,6). The dangling tuple (7.8) is padded to produce the tuple (`NULL`,7,8), a total of 6 tuples.

Problem 18: (c)

II is a straightforward implementation of the query, but I fails because it tries to extend a dotted path through the collection-valued expression `t.players`.

Problem 19: (c)

Again, II is a (fairly) straightforward way to express the query. The where-clause asks that there exist some player p in the set of players on team t whose name is Sue. If that condition is satisfied for team t, then the name of every player on the team is printed.

I fails because it makes the dual of the mistake from Problem 18(I): here there is an attempt to use a noncollection in a place (the from-clause) where a collection is required. That is, `p.playsFor` is not a collection and thus cannot be used where it appears in query I.

Problem 20: (d)

The only use for a relation whose attributes are a subset of some other relation's is in preserving information about entities or objects that otherwise would not appear in the relation with the larger schema. Choice (d) is an expression of this possibility. Note that for (a), while the premise is true (``name'' has a different meaning in the two relations), the implied conclusion doesn't follow, because `Teams.name` and `Players.teamName` are really the same attribute.

Problem 21: (c)

Yes, each attribute is a key by itself. But *BC* is therefore not a key. It is a superkey, however.

Problem 22: (a)

Each tuple of R at least matches itself, so there is a minimum of *n* tuples in the result. However, it is possible that all *n* tuples of R are identical, in which case there are *n^2* tuples in the result. Choice (a) captures these extremes.

Problem 23: (d)

Since each tuple of R and S can produce at most one tuple in the result, min(r,s) is a maximum output size. Remember that intersection automatically eliminates duplicates. Since R and S could have no tuples in common, 0 is the only lower bound, and choice (d) captures these extremes.

Problem 24: (b)

Here, you had to understand the meaning and implications of ``for each row.'' The deletion statement deletes two tuples --- both of the EE employees. Each time a tuple is deleted, the trigger is triggered, and since the when-condition is surely satisfied, it gives the Math person a raise. Thus, at the end, there is the CS person with a well-deserved $65,000 salary and the Math person overpaid at $54,000.

Problem 25: (b)

Choice (b) is safe; all variables appear in nonnegated, ordinary subgoals. Choice (a) is not safe because y appears only in the head and a negated subgoal, while (c) is not safe because y appears in the head, a negated subgoal, and an arithmetic subgoal.

Problem 26: (a)

Choice (a) is a straightforward translation of the Datalog rule. Choice (b) is nonsense; the renaming doesn't preserve the number of attributes. Choice (c) is also nonsense; *z* has no meaning in the relational algebra expression.

Problem 27: (b)

First, note that in both I and II, T(x,y) correctly computes PROJ_AB(Q JOIN R). However, I intersects T with P, so it is not correct. II correctly takes the union of T and P. III is also correct; it takes the union of P with the project-join expression directly, without the intermediate T. Thus, the answer is (b): II and III only.

Problem 28: (d)

This problem was intentionally hard, asking you to extrapolate something we did in class and tempting you with choice (c), which seems to relate to the work we did on preserving FD's in a decomposition, but doesn't answer the question. Regarding (c): true, this choice straightforwardly embeds the FD's. But that is not enough for a lossless join unless one of the decomposed relations also includes a key. In this case, the only key is AB, so we would have to add AB to be sure of a lossless join.

The easiest way to verify that (d) has a lossless join is first to consider the join of AB and ACE. Since A->CE, our theorem about a lossless join if the common attributes functionally determines one side or the other, tells us that the decomposition of ABCE into AB and ACE is lossless. Now, consider the join of ABCE with BCD. The intersection, BC, functionally determines D, so this join is also lossless. Now, we know that the join of AB, BCD, and ACE is lossless, so if you have faith in your reasoning you just mark (d) and go on to the next question.

However, if you want the sanity check of making sure that none of the other choices could be right, there is a trick that generalizes the argument we used to prove that theorem about when the join of two relations is lossless. Start with a tuple that appears in the result of the join, say *abcde*. Argue that there must be tuples in each of the decomposed relations that agree with it in appropriate attributes and have some other values elsewhere. Then use the FD's to try to infer that certain unknown symbols must be equal and eventually to infer that one of these tuples is really *abcde*. If you cannot prove that, after applying all FD's in all possible ways, then you have a counterexample. That is, you have a set of tuples that could be the original relation, and when you project it onto the decomposed relations, and rejoin, you get a tuple *abcde* that you did not have before.

Let's see how this applies to choice (c); you can try it with (a) and (b) yourself. The three tuples in relation ABCDE whose projections onto AC, CE, and BCD yields *abcde* in the join must look like:

A	B	C	D	E
a	b'	c	d'	e'
a'	b''	c	d''	e
a''	b	c	d	e''

Since all three tuples agree in C, and we have FD *C->E*, we can infer that all of *e*, *e'*, and *e''* are really *e*. However, this is all we can infer. Thus, the following relation instance:

A	B	C	D	E
a	b'	c	d'	e
a'	b''	c	d''	e
a''	b	c	d	e

is an example where projecting onto AC, CE, and BCD and then joining gets you something you didn't start with, namely *abcde*.

Problems 29: (d) and 30: (b)

II is essentially the join of three copies of `Family`, so it is easy to do in either Datalog or relational algebra. III is a recursive query, the canonical example of what you can do in Datalog but not relational algebra. I is an example of a standard ``trick'' we've used several times to simulate min or max in algebra or Datalog. You start by expressing (in either notation) the set of (p,c1,c2) such that p is a parent of both c1 and c2, and c1 was born before c2. If we project this set onto p and c2, we get the set of parent-child pairs such that the child is *not* the youngest of the parent. If we subtract this set from the set of all parent-child pairs, we get the set of parents and their youngest children. You should be able to follow this plan to express the query in

either Datalog or algebra.

Problem 31: (c)

In order for a rule of this form to be invalid, the first FD would have to be true, and the second false. In (a), the first FD, *AB->C* is false, so this cannot be the answer. In (b), the second FD is true, so that cannot be the answer. However, in (c), the first FD is trivially true (no two tuples agree on both B and C), while the second is false (the first and last tuples agree on C but disagree on A).

Problem 32: (a)

Choice (a) is correct; the first relation comes from *Cities* and the second from *States*. Note that *In* does not produce any relation because of the special rule for weak entity sets.

Problem 33: (a)

Choice (a) is the straightforward representation of this binary relationship. Choice (b) is not as good, because it introduces an extra concept that we do not need. Choice (c) is doubletalk. True, there is no need for a key in an ODL description, but *In* is an important concept (what state is a city in?) that deserves to be represented, as it is in the E/R diagram. Choice (d) is not good, because we would then have two representations of states: the attribute and the class.