Kevin Sass

11/4/19

CSC\_355\_401\_1040

Database Systems

Assignment 5

1. **A.**   
     
   1. A+ -> A
2. B+ -> B
3. C+ -> C
4. D+ -> AC
5. AB+ -> AB
6. AC+ -> AC
7. AD+ -> ACD
8. BC+ -> ABC
9. BD+ -> ABCD
10. CD+ -> ACD
11. ABC+ -> ABC
12. ABD+ -> ABCD
13. ACD+ -> ACD
14. BCD+ -> ABCD
15. ABCD+ -> ABCD

**B.** The superkeys are BD, ABD, BCD, and ABCD

**C.** The candidate key is BD

**2.** Going through the functional dependencies, we determine the first FD to not be in BCNF since the determinant “ContractID” by itself is not a superkey (you can’t get to Rate from ContractID alone). We eliminate the first FD from CONTRACT and form CONTRACT into 2 new relations: R1(ContractID, Type, Rate) and R2(ContractID, StartDate, Discount). To determine if these 2 relations have the lossless join property we will use the Binary Lossless Join Test. The common attribute in both relations is ContractID, so we have to determine if ContractID determines Type and Rate, OR ContractID determines StartDate and Discount. Looking at the starting functional dependencies, the first FD in fact states that ContractID determines StartDate and Discount, so the Binary Lossless Join Test returns True.

**3.**  **a.** The projections of F on R1, R2, and R3 are as follows:

R1: CustomerID -> FirstName  
 R2: CustomerID -> LastName  
 R3: CustomerID, Plan -> Amount

The decomposition does not have the dependency preservation property. First we have to see if we can derive F from the projections. The first one is true: CustomerID, Plan -> FirstName, LastName, Amount, since CustomerID determines both FirstName and LastName in the projections, and CustomerID + Plan determine the amount in the projections, so these are equivalent. However, the 2nd part of F (CustomerID -> FirstName, LastName, Plan) can’t be derived from the projections (We can’t determine Plan if we only have CustomerID). Therefore, The decomposition is not equivalent to the beginning relation.

**b.** The decomposition does have the lossless join property. We make a matrix with column attributes CID, FN, LN, P, and A in that order. We make the rows R1, R2, and R3 in that order. We look at the FD Projections and fill in ai for each attribute in each relation, and fill in the rest of the blank squares with bij. The first row ends up being a1, a2, b13, b14, b15 in that order. 2nd row is a1, b22, a3, b24, b25 in that order. Final row is a1, b32, b33, a4, a5. We then go thru the projections and first look at the determinants. CID is the determinant of R1, so we look in the table to see if CID has all rows with a1 filled in, and it does! That means we can now look at the 2nd half of the projection of R1 (FN), look for FN in the table and replace all rows in that column with a2. Next we go to the next projection R2 and it’s determinant is CID, in which all rows of CID in the table match, so we can look at the 2nd half of R2 in the table and turn all those LN values to a3. After we do this, we now have R3 showing a’s in every column, thus proving that the decomposition does in fact have the lossless join property.

**4. a.** First, the FD’s start off being A -> N; R -> T, A, N; and D, R -> P, T. First we look at the FD’s who’s right hand side have more than one attribute and separate those into single attributes. R -> T, A, N becomes R -> T; R -> A; and R -> N. D, R -> P, T becomes D, R -> P and D, R -> T. Next step we look for the FD’s whose left hand side is more than one attribute, split them into new single attribute FD’s, then see which of those new FD’s can replace the old one while still keeping the set equivalent. For D,R -> P, we can use D -> P or R -> P. However, none of those will work, so we end up keeping this double determinant FD. We move on to the next double determinant FD which is D,R -> T and split it into D -> T and R -> T and see which of these 2 can replace the old FD while still keeping the entire set equivalent. For D -> T, if we start with D, we can’t get to T so that doesn’t work. However, for R -> T, if we start with R, we can get to T already, so replacing this would leave the set equivalent. The last step is to check each of these functional dependencies in turn and ask if we get rid of this and drop it out of the set, can we derive it from a combination of the things that remain? If yes, we can take it out. If no, we leave it. First is A -> N, which when deleted, we cannot get from A to N, so we leave this in. Next is R -> T, which when taken out, we can still get to T from R from the last/bottom FD, so we can delete this. Now R -> A, we can’t get to A from R in any other way so we have to keep this one. R -> N: we can do R -> A, then A -> N and that gives us R -> N, so we can delete this one. D,R -> P: we can’t get P any other way so we keep this. Finally, R -> T is the only way to get to T so we keep it. The final minimal basis is A -> N, R -> A, D,R -> P, and R -> T.

**b.** First we take our minimal basis answer from a. and group it up into relations based on the left side. We get R1(A,N); R2(R,A,T); and R3(D,R,P). The third step is to check if one of these contains the candidate key (D,R), in which yes, R3 contains D and R. The fourth and final step: are any of these relations redundant, or reoccurring within another’s relation? The answer is no. The final 3NF relations are: R1(A,N); R2(R,A,T); and R3(D,R,P).