University of California, Los Angeles
Department of Computer Science

Computer Science 143 **Prof. Ryan Rosario**

Homework 5

Solutions

Part 1: A Purely Transactional Relationship

1. We will use the following transaction schedule S for this problem.

T_1	T_2
`read(A)`	
`write(A)`	
	`read(A)`
	`read(B)`
`read(B)`	
`write(B)`	

(a) Is S serial?

Two transactions are serial is one transaction completely finishes before the other one begins. That is not the case here, so the schedule is **not** serial.

(b) Is S conflict serializable? If so, what are the equivalent serial schedules and what is the proper ordering? Show your work using the swap method or the graph method.

We will create the dependency graph. A `write(A)` in T_1 appears before a `read(A)` in T_2, so there will be an edge $T_1 \rightarrow T_2$. A `read(B)` in T_2 occurs before a `write(B)` in T_1, so there will be an edge $T_2 \rightarrow T_1$. We do not have any `write`/`write` conflicts. The precedence graph looks as follows:

Since the precedence graph has a cycle, S is not conflict serializable.

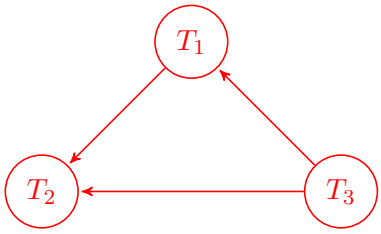
2. Look carefully at these three transactions T_1, T_2, T_3 and T_4.

T_1	T_2	T_3
`write(A)`		
		`write(B)`
		`write(B)`
	`write(A)`	
		`read(B)`
	`read(B)`	
	`read(A)`	
`read(B)`		

(a) Construct the precedence graph for this schedule S.

`write(A)` in T_1 occurs before a `write(A)` in T_2, thus $T_1 \to T_2$.
`write(B)` never occurs before another `write(B)` in this schedule.
`write(A)` in T_1 occurs before a `read(A)` in T_2, thus $T_1 \to T_2$.
`write(B)` in T_3 is followed by a `read(B)` in T_1 and T_2, thus $T_3 \to T_1$ and $T_3 \to T_2$.
`read(A)` never occurs before a `write(A)` in this schedule.
`read(B)` never occurs before a `write(B)` in this schedule.

The precedence graph looks as follows:



(b) Is S conflict serializable? Justify your answer. If yes, compute the a serial ordering of the transactions.

Yes, there is no cycle in the precedence graph, so the schedule is conflict serializable and we can compute a serial ordering of the transactions using topological sort. See the algorithm from the lecture. One ordering is $T_3 \to T_1 \to T_2$. In general, topological sort may result in different orderings all of which are correct, depending on how nodes are ordered when being inserted into the queue (for the BFS/Kahn's version of topological sort. The same is true for the DFS version, except orderings may be differ based on how nodes are ordered when pushed onto the stack.

Part 2: There's Nothing Wrong with Being Abnormal Unless you are a Relation

1. Suppose that we decompose the schema $R(A, B, C, D, E, F)$ into $R_1 = (A, B, C, F)$ and $R_2 = (A, D, E)$. Given the following functional dependencies hold, is the decomposition lossless? Explain your answer.

$$F = \{A \to B, A \to C, CD \to E, B \to D, E \to A\}$$

Note that the decomposition need not be in BCNF. There are a variety of ways to determine if a decomposition is lossless or not. If we had the data for these relations and subrelations, we could check if $R_1 \bowtie R_2 = R$. Since we do not, we must use other methods. There are three ways to determine if a decomposition is lossless:

(1) Using the definition from the book
   (a) via the functional dependency closure F^+.
   (b) via the attribute closure.
(2) the chase

Method 1: Definition of Lossless

We must check that three conditions are true:

(a) $R_1 \cup R_2 = R$
(b) $R_1 \cap R_2 \neq \{\}$
(c) $R_1 \cap R_2$ is a superkey for either R_1 **or** R_2.

Note that the third condition is equivalent to checking if $R_1 \cap R_2 \to R_1$ or $R_1 \cap R_2 \to R_2$ exists in F^+. In other words, if one of those functional dependencies can be derived by applying Armstrong's Axioms to the functional dependencies in F.

We will check the first two conditions first:

$$R_1 \cup R_2 = (A, B, C, F) \cup (A, D, E) = (A, B, C, D, E, F) = R$$

The first condition is satisfied. Now let's check the second one:

$$R_1 \cap R_2 = (A, B, C, F) \cap (A, D, E) = A \neq \{\}$$

The second condition is also satisfied. Now we check the third condition.

Method 1A: Definition of Losslessness using Superkeys

This method uses the attribute closure to determine if $R_1 \cap R_2$ is a superkey for R_1 or R_2. This is my preferred method because it is a very easy plug-and-chug algorithm, whereas trying to deduce equivalent functional dependencies using Armstrong's Axioms can sometimes be error prone.

$R_1 \cap R_2 = A$, so we find the attribute closure for A, denoted A^+. We can notice right away that we will never, ever, derive F because there is no functional dependency involving F. Therefore, we can declare that A is not a superkey for R_1. We have not yet satisfied the third condition, so let's look at R_2 and check if A is a superkey for R_2.

We initialize our closure with A: `result = A` and start the first iteration, trying to see what we can infer from each functional dependency in F:

(a) Try $A \to B$. A is in the result set, so we can use $A \to B$ to infer B. We add B to the closure, which is now `result = AB`.

(b) Try $A \to C$. A is in the result set, so we can use $A \to C$ to infer C. We add C to the closure, which is now `result=ABC`.

(c) Try $CD \to E$. CD is not in the result set, so we cannot do anything.

(d) Try $B \to D$. B is in the result set, so we can use $B \to D$ to infer D. We add D to the closure, which is now `ABCD`.

(e) Try $E \to A$. E is not in the result set, so we cannot do anything.

At this point, we have completed one iteration of this algorithm, and we have updated the closure, so we must start a second iteration and start with the first functional dependency again.

(a) Try $A \to B$. Although A is in the result set, B is also already in the result set so we move on.

(b) Try $A \to C$. Although A is in the result set, C also is, so we move on.

(c) Try $CD \to E$. CD is **now in the result set!** We use $CD \to E$ to derive E and add it to the result set. The closure is now `result = ABCDE`.

(d) Try $E \to A$. E is in the result set, but so is A, so we do not change anything.

In this second iteration, we updated the closure, so technically we must do another iteration; however, since we have gotten this far, we can stop since it is pointless because we have already functionally determined all attributes of R_2. But for good measure, we will show another iteration.

(a) Try $A \to B$. Although A is in the result set, B is also already in the result set so we move on. No change.

(b) Try $A \to C$. Although A is in the result set, C also is, so we move on. No change.

(c) Try $CD \to E$. Although CD is in the result set, E also is, so we move on. No change.

(d) Try $B \to D$. Although B is in the result set, D also is, so we move on. No change.

(e) Try $E \to A$. Although E is in the result set, E also is, so we move on. No change.

We finished this iteration without updating the closure result, so we STOP.

We conclude that $A^+ = \{A, B, C, D, E\}$. That is, we can infer B, C, D and E from A. Since $R_2 = (A, D, E)$ and all of R_2's attributes are in A^+, that is, $R_2 \subseteq A^+$, A is a superkey for R_2 and the third condition of losslessness is true. Since all three conditions hold, **yes, the decomposition is lossless.** Note that in class, our examples of attribute closure were *exactly equal to* R, and we do not need to be that strict it just worked out for those examples.

Switching back to R_1 for a moment... Since $A^+ = R_1 \setminus F$, A is not a superkey for R_1, but AF **is** (you can prove this by finding $\{AF\}^+$.

Method 1A: Definition of Losslessness with F^+

With this method, we check that $R_1 \cap R_2 \to R_1$ or $R_1 \cap R_2 \to R_2$ can be derived from applying Armstrong's Axioms to F. That is, we are checking that these functional dependencies are in the closure F^+. For this particular problem, we are checking that $A \to R_1$, that is $A \to ABCF$. So we must show that $A \to A$, $A \to B$, $A \to C$, and $A \to F$ on R_1. Because the attribute F does not appear in any functional dependencies, we can never, ever, derive $A \to F$, so we switch to R_2.

Now we check that $A \to R_2$, that is, $A \to ADE$. We must show that $A \to A, A \to D, A \to E$. $A \to A$ is trivial. By the union property, we can use $A \to B$ and $A \to C$ to infer $A \to BC$. From $C \to D$, we can infer $A \to CD$. By the decomposition rule, we get $A \to C$ (which we already were given) and $A \to D$. We can then use $CD \to E$ to infer E. We were able to derive A, D, and E, therefore $A \to R_2$ holds and the third condition also holds, thus **yes, the decomposition is lossless.**

Method 2: The Chase

Another popular method that is discussed in many other resources, including textbooks, just not ours, is the chase method. You can read about the chase algorithm at `https://en.wikipedia.org/wiki/Chase_(algorithm)`.

We start with a tableau containing a column for each attribute in R and a row for each of the decomposed relations. So we have 6 columns (A thru F) and two rows (R_1 and R_2). In each row, we list each attribute as a variable, and any attributes that do not appear in a subrelation will have a subscript added to it. We then cycle through the set of functional dependencies. Recall that a functional dependency $F : X \to Y$ means that for two tuples t_1 and t_2, if $t_1[x] = t_2[x]$ then $t_1[y] = t_2[y]$. We will use this fact in this process. We want to show that the decomposition is lossless, that is, every tuple in R is preserved in the natural join $R_1 \bowtie R_2$ and we do not get any additional tuples. That is, we want to check that $t = (a, b, c, d, e, f)$ exists $\forall t \in R$ and we can do this by checking each functional dependency against our tableau.

Our initial tableau looks as follows with rows representing R_1 and R_2 respectively:

A	B	C	D	E	F
a	b	c	d_1	e_1	f
a	b_2	c_2	d	e	f_2

First, we apply $A \to B$ to both subrelations. We see (a, b) in R_1 and (a, b_2) in R_2. These **must** be equal for $A \to B$ to hold, so we change $(a, b_2) \to (a, b)$.

A	B	C	D	E	F
a	b	c	d_1	e_1	f
a	b	c_2	d	e	f_2

Next we apply $A \to C$ to both subrelations and set (a, c_2) in R_2 to (a, c).

A	B	C	D	E	F
a	b	c	d_1	e_1	f
a	b	c	d	e	f_2

Next we apply $CD \to E$. By similar reasoning, we know that $d_1 = d$ and $e_1 = e$ otherwise $CD \to E$ wouldn't hold.

A	B	C	D	E	F
a	b	c	d	e	f
a	b	c	d	e	f_2

Note that we have $t = (a, b, c, d, e, f)$ thus we STOP and declare **the decomposition is lossless**. For good measure, let's look at the other two functional dependencies.

We apply $B \to D$. We don't make any changes to the tableau because we already see that $b = b, d = d$. Finally, we check $E \to A$. Again, we don't make any changes to the tableau because we already see that $e = e, a = a$.

2. List non-trivial functional dependencies satisfied by the following relation. You do not need to find *all* dependencies. In other words, please find F, but there is no need to find F^+. It is enough to identify a set F of functional dependencies that imply all functional dependencies satisfied by this relation (i.e. irreducible).

A	B	C
a_1	b_1	c_2
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Note that every $a_i \in A$ maps to exactly one $\beta_j \in B$, **so one functional dependency is** $A \to B$**.** $A \to C$ does not hold because a_2 maps to both c_1 and c_3. $B \to C$ does not hold because b_1 maps to c_1 and c_3. $C \to A$ and $C \to B$ also holds.

$BC \to A$ also holds, but it is not a basis dependency because we can derive it from $C \to A$ first via augmentation to get $BC \to AC$ and then to the set $BC \to A$ and $BC \to C$ via decomposition, where $BC \to C$ is technically a trivial dependency since $C \in BC$.

In other words, you are asked to find F, but not F^+.

3. Assume the following set of functional dependencies hold for the relation $R(A, B, C, D, E, G) : F = \{A \to B, A \to C, C \to E, B \to D\}$

   Is R in Boyce-Codd Normal Form (BCNF)? Explain your answer. If it is not, normalize it into a set of relations in BCNF such that the decomposition is lossless and determine whether or not it is dependency preserving. If it is not, which dependencies are not preserved?

   We determine if a R is in BCNF by considering all function dependencies in $\mathcal{F}$ and verifying that **each** of them is either

   (a) trivial ($\beta \subseteq \alpha$, $\alpha \cup \beta = R$); or

   (b) α is a superkey.

   First we must determine the key. A looks like a good choice so let's compute A^+. By $A \to BC$ we get $BC \in A^+$. By $C \to E$ and $B \to D$ we get that $DE \in A^+$, so $A^+ = \{ABCDE\}$.

   But wait a minute!

   A can't be a key because it doesn't determine G! So what if we take AG to be the key? That's what we do. AG is a superkey because $(AG)^+ = R$.

Now, α, the lefthand side of each functional dependency, must be a superkey for R to be in BCNF. Right off the bat, A from $A \to BC$ is not a superkey, so $\boxed{R \text{ is not in BCNF.}}$

The order in which we decompose matters, in the sense that we may not get a dependency preserving decomposition. The problem only asks to decompose to BCNF though. This happens if we start with $A \to BC$. Consider $C \to E$ instead.

We decompose into

$$R_1 = (C, E), R_2 = (A, B, C, D, G)$$

Now we are done with $C \to E$. Then we use $B \to D$ to get

$$R_1 = (C, E), R_2 = (B, D), R_3 = (A, B, C, G)$$

Now we are done with $B \to D$. Now we deal with $A \to BC$ to get

$$\boxed{R_1 = (C, E), R_2 = (B, D), R_3 = (A, B, C), R_4 = (A, G)}$$

Note that if we continue by using the decomposition rule on $A \to BC$ to get $A \to B$ and $A \to C$, we over-normalize, and end up losing the functional dependency $A \to BC$. This would not be correct anyway, because recall that when we decompose using FDs $X \to Y$ that violate BCNF, we split the (sub)relation into two pieces: $R_i = X^+$ and $R_j = X \cup (R \setminus X^+)$.

In this particular problem, the order (there are $3! = 6$ of them if we consider $A \to BC$) does not matter and we get the same result with each. This will not always happen.

4. Consider the following relational schema describing musical events in California in some prior decade. Assume each event has at least one band. Bands stick to one genre of music and they do not visit the same venue twice in the same year. This implies that you should consider the full context of the problem and not just this instance of the data.

A	B	C	D
Venue	Year	Band	Genre
Hollywood Bowl	1999	Mighty Mighty Bosstones	ska
Royce Hall	1999	Mighty Mighty Bosstones	ska
Shoreline Amphitheater	2001	Mighty Mighty Bosstones	ska
Shoreline Amphitheater	2001	Dinosaur Jr.	rock

Is this relation in 2NF? 3NF? Determine the functional dependencies and keys and justify your answer.

This problem caused frustration for some, likely because students were trying to construct functional dependencies from only the constraints I gave and ignored the concept of keys. The fact is, we need to find a candidate key and this proves challenging using just the constraints. Remember that a candidate key is a minimal superkey. So let's start by finding all superkeys starting with single attributes and moving up to those containing all attributes. We can then use the constraints to whittle them down.

There are no single attribute candidate keys since there are no single attribute superkeys. There are 6 possible two-attribute sets. Using the table, we can eliminate AB, AC. AD does not make intuitive sense since by the context of the problem, multiple genres can be performed at each venue. BC, BD can be eliminated using the table. CD can also be eliminated as a possible superkey by the table, but per the provided constraint, we have $C \to D$. There are 4 possible three-attribute sets. ABC is a possible contender

since each band can only be associated with one genre. ABD is eliminated by the table. ACD wouldn't make sense because that implies a band cannot return in another year (context). BCD is eliminated by the table. $ABCD$ is a trivial superkey. We only have two superkeys on this relation, ABC and $ABCD$ but ABC is the minimal superkey, and thus it must be the candidate key as well. Thus we can infer the functional dependency $ABC \to D$.

Our functional dependencies are $ABC \to D$ and $C \to D$.

We have a key, there are no implied nulls in this context, all attributes are atomic types, and we do not have repeated groups, thus the the relation is in at least First Normal Form (1NF). To be in Second Normal Form (2NF) all attributes must either be in a candidate key, or determined by the **entire** candidate key. D is functionally determined by C, and C is is contained in the candidate key. Thus, D is *partially* dependent on the candidate key, which means $C \to D$ is a violation of 2NF.

The relation is not in 2NF, thus it cannot be in 3NF.

5. We now modify the schema to include the number of attendees. The same constraints from the previous part apply:

A	B	C	D
Venue	Year	Band	Attendees
Hollywood Bowl	1999	Mighty Mighty Bosstones	10000
Royce Hall	1999	Mighty Mighty Bosstones	8000
Shoreline Amphitheater	2001	Mighty Mighty Bosstones	90000
Shoreline Amphitheater	2001	Dinosaur Jr.	10000

Is the new schema in 2NF? 3NF? BCNF? Explain your answer.

The only non-key attribute is D, `Attendees`, which depends on the entire key, so it is indeed in 2NF. `Attendees` depends on only the key and not another non-key attribute, thus the relation is also in 3NF.

We still have $ABC \to D$ with ABC as the candidate key. Each lefthand side of the functional dependencies is a superkey, thus **this relation is in BCNF so it is also in 3NF and 2NF**. Note that this relation does not include the genre attribute. Since constraints were given in Problem 4, we are finding FDs that hold on the entire relation (context) and not just this little table.

Important: On the exam, I will make it very clear if we are referring to context or this instance.