Lecture 11 — Decomposition: Functional-Dependency Theory

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 13, 2017

Seeking Closure

Earlier we talked about the closure of a set of functional dependencies.

Recall from earlier that the closure contains all the functional dependencies that are explicitly satisfied as well as those that are logically implied.

That is, if $A \rightarrow B$ and $B \rightarrow C$ then it is implied that $A \rightarrow C$.

If the functional dependencies are $A \rightarrow B$ and $B \rightarrow C$ then we know that for two tuples t_1 and t_2 if $t_1[A] = t_2[A]$ then $t_1[B] = t_2[B]$ and $t_1[C] = t_2[C]$.

The notation to show the closure of a set F of functional dependencies is F^+ as previously discussed.

If F is large there are many rules and many implied rules and we have to construct every logically implied element of F^+ from first principles.

Instead, we would like to use *axioms*, handy rules of inference, that allow us to reason about the dependencies in a simpler way.

Armstrong's Axioms

The first three axions are simple enough and are called Armstrong's Axioms:

- **Reflexivity**: If α is a set of attributes and β is contained within α, then $\alpha \rightarrow \beta$ holds.
- **Augmentation**: If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity**: If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Some convenient shortcuts:

- **Union**: If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition**: If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (reverse of previous rule).
- **Pseudotransitivity**: If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Functional Dependency Theory Example

The relation r has the attributes (A, B, C, G, H, I).

The functional dependencies are: (1) $A \rightarrow B$, (2) $A \rightarrow C$, (3) $CG \rightarrow H$, (4) $CH \rightarrow I$, (5) $B \rightarrow H$.

Based on the rules that we have, what logically implied functional dependencies can we observe?

Functional Dependency Theory Example

There are three:

- $A \rightarrow H$ which is found by transitivity ($A \rightarrow B$ and $B \rightarrow A$).
- $CG \rightarrow HI$ which is found by the union rule ($CG \rightarrow H$ and $CG \rightarrow I$).
- $AG \rightarrow I$ which is found by pseudotransitivity ($A \rightarrow C$ and $CG \rightarrow I$).

Computing the Closure

1. The initial condition is that F^+ begins as F.
2. For each functional dependency f in F+ F^+:
   1. apply the transitivity rule to f and add it to F^+
   2. apply the augmentation rule to f and add it to F^+
3. For each pair of functional dependencies f_1 and f_2, if they can be combined using transitivity, add the newly created combination to F^+.
4. If anything was added in steps 2 or 3, go back to step 2; otherwise the algorithm terminates.

Suppose we have some attribute(s) α and we wish to determine if it is a superkey.

The strategy we will use requires us to compute the set of attributes that are functionally determined by α.

An attribute B is functionally determined by α if $\alpha \rightarrow B$.

Then the simple algorithm requires us to compute F^+...

More Efficient Algorithm

A more efficient algorithm:

1. The initial condition is that α^+ begins as α.
2. For each functional dependency $\beta \rightarrow \gamma$ in *F*, if β is contained in α^+, then γ is added to α^+
3. If anything was added in step 2, repeat step 2; if nothing was added, the algorithm terminates.

Much like computing the closure of *F*, here we compute the closure of α.

Back to the Example

Given the rules, is *A* a superkey? Let's go through the steps.

The initial condition is that *A* is in α^+ (the result).

The result is $ABCH$.

This is not the full set ($ABCGHI$) and therefore we conclude that A is not a superkey.

Back to the Example

This procedure can be repeated for any individual attribute and we will quickly find that no single attribute is a superkey for this relation.

This is perhaps somewhat obvious from our list of functional dependencies.

If we look carefully at them we can notice that neither A nor G ever appears on the right hand side of any of the functional dependencies.

All the other attributes do.

Back to the Example

That gives us a hint that a superkey might be AG, and that assumption can of course be tested by following the rules as above.

If our candidate is AG as suggested, we start off by saying the result is AG.

We will, in fact, find that AG is a superkey.

Use of Attribute Closure

There are three ways we can use this attribute closure algorithm:

- As above, to test if α is a superkey.
- Check if a functional dependency holds (by determining if it is in α^+.
- As another way to compute F^+; we just compute the α^+ for each α and then combine them.

Suppose we have a set of functional dependencies F on a relation.

Whenever a user wants to update the data inside this relation, the database system must check that all functional dependencies in F are still satisfied.

We would like to simplify the set to a minimal number of rules that has the same closure.

For example, if the rules say $A \rightarrow B$ and $B \rightarrow C$ and $A \rightarrow C$ we can immediately identify that there is redundancy.

We could remove the rule $A \rightarrow C$.

You could argue that this sort of thing should not be necessary: designers should not introduce redundant rules and if they do it is their own fault...

Formally speaking, an attribute of a functional dependency is extraneous (unnecessary) under the following two scenarios:

- A is extraneous in α if A is in α and F logically implies $(F - (\alpha \to \beta)) \cup ((\alpha - A) \to \beta))$.
- A is extraneous in β if A is in β and the set of functional dependencies $(F - (\alpha \to \beta)) \cup (\alpha \to (\beta - A))$ logically implies F.

Extraneous Dependencies

To check if A is extraneous on the right hand side, the routine is fairly simple.

Remove A from the right hand side (β) from the rule that it is in (e.g., if the rule is $D \rightarrow AB$, replace it with $D \rightarrow B$).

Then compute the canonical cover of α (in this example, D). If the canonical cover included A, then A was extraneous in β.

Extraneous Dependencies

To check if A is extraneous on the left hand side:

Remove A from the left hand side (α) and check if this reduced set still functionally determines β.

So if the rule is $DF \rightarrow GH$, remove F and we are left with the uncertain rule $D \rightarrow GH$, which we need to test to be sure.

We compute the closure of D without this modified rule and see if it includes GH.

If it does, then F was extraneous in the left hand side.

The canonical cover for F is denoted F_c and it is a minimal representation of F.

It requires two rules: (1) no functional dependency in F_c has an extraneous attribute, and (2) the left side of each functional dependency is unique.

Canonical Cover Example

The schema is simple (A, B, C) and our functional dependencies are $(A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C)$.

Well, anyway, it is simple enough, we can use the union rule to combine the rule $A \rightarrow BC$ and $A \rightarrow B$ (pretty obvious).

And then we can remove extraneous attributes to get this down to $A \rightarrow B$ and $B \rightarrow C$.

This is slightly redundant and we can figure out by transitivity that $A \rightarrow BC$.

Canonical Cover Example 2

Another example: our schema has more attributes and the functional dependencies are $(B \rightarrow A, D \rightarrow A, AB \rightarrow D)$.

In this case, all the left hand sides are different, so we can't combine any that way.

The right hand sides are all single attributes so we don't need to look for extraneous attributes there.

But we should look at the left hand side.

Canonical Cover Example 2

The only one that is not minimal there is the functional dependency $AB \rightarrow D$ and we would like to know if we can eliminate A or B from the left side.

Looking at the rule that says $B \rightarrow A$ we can probably reason pretty well that if $B \rightarrow A$ then $B \rightarrow AB$.

So we can replace the one double-left-side rule with $B \rightarrow D$.

Then we have a set that is equivalent: $(B \rightarrow A, D \rightarrow A, B \rightarrow D)$.

There is some redundancy here, though, and we could find out that $B \rightarrow A$ can be eliminated, leaving us with $(B \rightarrow D, D \rightarrow A)$.

Because canonical cover is about finding a minimal equivalent set, it is not necessarily that there is only one correct answer.

If there is redundancy, we may need to delete one of two attributes (but not both).

Let's go back to the example from earlier with the three attributes. Our functional dependencies are still ($A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C$).

If we test $A \rightarrow BC$ we will find that both B and C are extraneous.

One person might choose to delete B and another person might choose to delete C. Neither choice is wrong.

Decomposition is breaking up a relation into two or more smaller ones.

The motivations for doing so are something we will come back to later on, but for the moment, just assume there are good reasons.

A decomposition is lossless if no information is lost by splitting a relation r into smaller relations r_1 and r_2.

If information is loss it is called lossy (and that is undesirable).

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

If the decomposition is lossless then every functional dependency f in F holds in spite of the fact that the relation is split.

But we can't really cheat and just "forget" to put in the functional dependencies to make it work, now can we?

There needs to be some attribute or attributes that link together the two tables.

One relation needs to have some way of referencing another.

The formal definition for a lossless decomposition says that one of the following must hold on the two relations:

$R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$.

That is to say, the intersection of the two relations most be sufficient to uniquely identify one of the two of them.

Lossless Decomposition Example

Suppose we have employee with *id, name, street, city, province, postalcode*.

We tried to split it into r_1 as *id, name*, and r_2 as *name, street, city, province, postcode*.

Is this going to work? The intersection of these two is *name* and this is not a superkey for either r_1 or r_2 because names are not unique.

Testing Lossless Decomposition

There is a formal way for testing that dependencies have been preserved.

There is a computationally expensive algorithm outlined in the textbook.

There's also one that is premised on the idea that if each functional dependency f can be tested on one relation only, then just test it on that relation.

Unfortunately that last assumption is not necessarily true; it could happen that a functional dependency is across tables...

Lossless Decomposition Algorithm

Our solution is then a modified algorithm that does not require us to compute the closure of F.

The algorithm is executed for each functional dependency $\alpha \rightarrow \beta$:

1. *result* = α
2. for reach relation R_i in the decomposition
    1. $result = result \cup ((result \cap R_i)^+ \cap R_i)$
    2. if *result* did not change at all, break out of the for loop

If *result* contains all attributes in β, then the dependency is preserved. And this must hold for all dependencies $\alpha \rightarrow \beta$.

The initial state is the relations that we have, and we also need to then compute F^+. Then the steps are:

```
result := {R};
done := false;
compute F+;
while (not done) do
    if (there is a schema Ri in result that is not in BCNF)
        then begin
                let α → β be a nontrivial functional dependency that holds
                on Ri such that α → Ri is not in F+, and α ∩ β = ∅;
                result := (result − Ri) ∪ (Ri − β) ∪ ( α, β);
            end
        else done := true;
```

The algorithm produces relations in BCNF and a lossless decomposition.

This is because when we replace a schema R_i with $(R_i - \beta)$ and (α, β) then the intersection of these two relations is α.

let F_c be a canonical cover for F;
$i := 0$;
for each functional dependency $\alpha \rightarrow \beta$ in F_c
    $i := i + 1$;
    $R_i := \alpha\,\beta$;
if none of the schemas R_j, $j = 1, 2, \ldots, i$ contains a candidate key for R
  then
    $i := i + 1$;
    $R_i :=$ any candidate key for R;
/* Optionally, remove redundant relations */
repeat
    if any schema R_j is contained in another schema R_k
      then
        /* Delete R_j */
        $R_j := R_i$;
        $i := i - 1$;
until no more R_js can be deleted
return $(R_1, R_2, \ldots, R_i)$