ECE356: Database Systems Winter 2018

Lecture 11 — Decomposition: Functional-Dependency Theory

Jeff Zarnett *2023-08-11*

Functional-Dependency Theory

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 $A \rightarrow C$ and this is true no matter how many steps are in between, because this is transitive. More formally, 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.

It is unlikely, however, that we want to compute F^+ from the definition of functional dependencies; 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. The first three axions are simple enough and are called *Armstrong's Axioms*, named after the person who came up with them [SKS11]:

- **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.

These axioms are considered both sound and complete, because they do not produce any errors and they allow generation of F^+ given F. There are proofs of these properties, but we prefer to omit those and just focus on using them. This is a minimal set, though, and there are some rules that we can derive some more from Armstrong's rules that would be convenient shortcuts [SKS11]:

- **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.

We'll work on the example in the textbook([SKS11]) which says that the relation r has the attributes (A, B, C, G, H, I) and the functional dependencies are: (1) $A \rightarrow B$, (2) $A \rightarrow C$, (3) $CG \rightarrow H$, (4) $CG \rightarrow I$, (5) $B \rightarrow H$.

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

There are five:

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

The full algorithm for building up the closure just uses Armstrong's rules and is as below. The algorithm terminates when there is nothing left to add, which is certain to happen because the worst case scenario is that everything is functionally dependent on everything else which would mean if there are n attributes, there are $2^n \times 2^n$ possible functional dependencies. Now, the algorithm [SKS11]:

1. The initial condition is that F^+ begins as F.
2. For each functional dependency f in F+ F^+:
   (a) apply the transitivity rule to f and add it to F^+
   (b) 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.

Closure of Attribute Sets. 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^+ as above and then take all functional dependencies where α is the left hand side and the union of the right hand side of all dependencies, but this is difficult and expensive because F^+ may be large. Instead, a more efficient algorithm follows [SKS11].

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.

Going back to the earlier example with the five functional dependencies, we would like to evaluate something to see if it is a superkey. Given the rules, is A a superkey? Let's go through the steps.

The initial condition is that A is in α^+ (the result). Based on the rules, we can look at the rule $A \rightarrow B$ and see that we can add to the result B, which is now AB. The next rule that says $A \rightarrow C$ allows us to add C to the result so ABC. The rules that require CG on the left hand side cannot be added, at least not yet, because CG does not appear in the result (C does, but not G). We can look at the rule for $B \rightarrow H$ and add H to the result meaning the result is $ABCH$. This is not the full set ($ABCGHI$) and therefore we conclude that A is not a superkey.

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, but all the other attributes do. 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. Then iterating through the functional dependencies: the first rule of $A \rightarrow B$ means we can add B to get ABG. The next rule of $A \rightarrow C$ lets us make the result $ABCG$. The dependency $CG \rightarrow H$ allows us to add H and get $ABCGH$, and the dependency $CG \rightarrow I$ means the result is $ABCGHI$ which encompasses all attributes and we can terminate the algorithm here; our guess based on the hints was a good one.

There are three ways we can use this attribute closure algorithm [SKS11]:

- 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.

Canonical Cover

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 (i.e., no key constraints or other rules are broken) [SKS11]. It is worth noting that in real life the rules are only enforced if you actually tell the database about them. If an update or insertion is inconsistent with the established rules, it is not permitted to happen.

Rather than checking every single functional dependency exactly as it is written, which may have some redundancy, we would like to simplify the set to a minimal number of rules that has the same closure. For example, if the rules say $A \to B$ and $B \to C$ and $A \to C$ we can immediately identify that there is redundancy and we could remove the rule $A \to C$ and still get the same closure, because that rule is implied and need not be checked separately.

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. That is nice, but we don't live in a perfect world. Database designers who sit down and work out functional dependencies in advance and draw diagrams and plan ahead will probably have few redundant rules. But that's not what happens a lot of the time, either the system is designed well and then grows to have duplicate rules over time, or, very commonly, people just create some tables without thinking about it much and "we'll fix it later if we have to" is the approach. But it's hard to fix later, and in the meantime, the database has done a lot of unnecessary work...

Formally speaking, an attribute of a functional dependency is *extraneous* (unnecessary) under the following two scenarios [SKS11]:

- 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.

The first scenario is about removing a redundant attribute A from the left hand side of an implication; the second is about removing it from the right hand side. An example of the first is if our rules say $AB \to C$ and $A \to C$, then we could remove B from the left hand side. For the other scenario, if $AB \to CD$ and $A \to C$ are the functional dependencies, then C is extraneous in the right hand side [SKS11].

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 $D \to AB$, replace it with $D \to B$) and then compute the canonical cover of α (in this example, D). If the canonical cover included A, then A was extraneous in β.

To check if A is extraneous on the left hand side, the idea is pretty much the same: remove A from the left hand side (α) and check if this reduced set still functionally determines β. So if the rule is $DF \to GH$, remove F and we are left with the uncertain rule $D \to GH$, which we need to test to be sure. We compute the closure of D without this modified rule and see if it includes GH (actually, all attributes in β). 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. The second rule means we can't have two rules that say $A \to B$ and $A \to C$ respectively; we must combine them in one single rule $A \to BC$. By making the minimal set, it makes it easier to test if the functional dependencies are satisfied.

Let's do an example from [SKS11]: 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 $A \rightarrow BC$. These rules are pretty easy to test and we prefer to test these two versus the original four. This is a pretty terrible schema, actually, because there are way too many dependencies and only three attributes.

Let's try instead a different one from [EN11]: 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. 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 $B \rightarrow AB$ (which you can also reason as adding B to both sides of the rule $B \rightarrow A$ becomes $BB \rightarrow AB$ and BB is just B). 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. This is normally not an issue, because we just test one of the two, and if it is redundant, we delete it and move on to testing other attributes in the modified set. But one person might choose to delete B and another person might choose to delete C. Neither choice is wrong, but they will produce two different canonical cover sets, both of which are equally correct. The more functional dependencies there are with redundancy to be eliminated, the more possible correct answers there are.

Lossless Decomposition

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 lost it is called *lossy* (and that is undesirable). The more precise definition of this is specified in relational algebra as $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$ (or in SQL, the natural join of the two tables with all attributes) [SKS11].

How can we tell if a join is lossless or lossy? You may have guessed just from the placement in the topics covered that this has something to do with the functional dependencies (meta gaming the lectures?) and that would be correct. 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?

Thinking carefully about the way the decomposition must work, there needs to be some attribute or attributes that link together the two tables. This does mean some data is duplicated. One relation needs to have some way of referencing another, something like an address that references a person meaning the address relation contains the identifier of the person.

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.

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

There is a formal way for testing that dependencies have been preserved. There is a computationally expensive algorithm outlined in the textbook, as well as 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 and we're done. Unfortunately that last assumption is not necessarily true; it could happen that a functional dependency is across tables...

Our solution is then a modified algorithm that does not require us to compute the closure of F. The algorithm is as follows, executed for each functional dependency $\alpha \rightarrow \beta$ [SKS11]:

1. $result = \alpha$
2. for reach relation R_i in the decomposition
   (a) $result = result \cup ((result \cap R_i)^+ \cap R_i)$
   (b) 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 $\alpha \rightarrow \beta$.

Which is explained in two key ideas as follows. The first idea is testing if each dependency $\alpha \rightarrow \beta$ is satisfied in F', where F' is the union of the *restrictions* of F. That is, when we take the part of F that can be applied to a relation R_i we get F_i and F' is the union of all F_is. We compute the closure of α under F' to see if that includes β. The second idea is to use a modified version of the attribute closure algorithm to compute closure under F' without actually computing F' which would be expensive. For full details, see [SKS11], but keep in mind this is polynomial time rather than exponential time like computing F^+ would be.

Algorithms for Decomposition

We have looked at very simple schemas (and probably will continue to look at very simple ones in this course). For those it is not too difficult to see how to get the schema into BCNF or 3NF or design it that way in the first place. But if we need to take a schema and decompose it, for a sufficiently complex schema, there are some algorithms that make this process consistent.

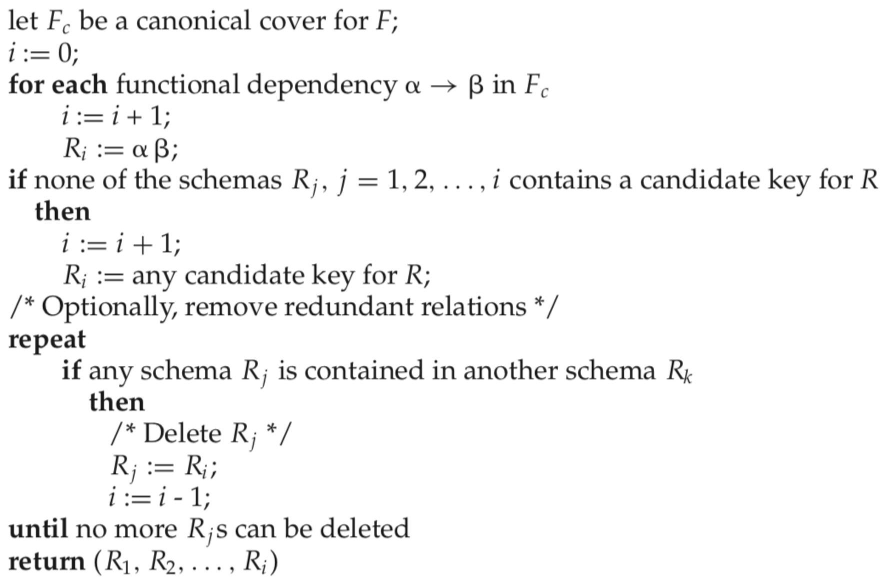
BCNF. To decompose to BCNF, the algorithm is as follows [SKS11]. The initial state is the relations that we have, and we also need to then compute F^+. Then the steps are:

```
result := {R};
done := false;
compute $F^+$;
while (not done) do
    if (there is a schema $R_i$ in result that is not in BCNF)
      then begin
              let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds
              on $R_i$ such that $\alpha \rightarrow R_i$ is not in $F^+$, and $\alpha \cap \beta = \emptyset$;
              result := (result $-$ $R_i$) $\cup$ ($R_i - \beta$) $\cup$ ( $\alpha, \beta$);
           end
      else done := true;
```

BCNF Decomposition algorithm in pseudocode [SKS11]

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.

3NF. If instead we wanted to put things in 3NF, it is a somewhat more complicated algorithm. It is also dependency preserving and lossless. It explicitly builds a schema for each dependency in a canonical cover, ensuring that each schema contains a candidate key. Unlike the BCNF decomposition algorithm, this builds the relations up rather than breaks them down. A formal proof of the algorithm below is in [SKS11]:



```
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 $R_j$s can be deleted
return ($R_1, R_2, \ldots, R_i$)
```

3NF Decomposition algorithm in pseudocode [SKS11]

References

[EN11] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 6th Edition*. Addison-Wesley, 2011.

[SKS11] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw Hill, 2011.