# CSCI 4333 Section 1 Design of DB Systems

## 4/12/2024 (self - annotation)

**Introduction to Functional Dependency and Normalization**

by K. Yue

**1. Introduction to Normalization**

* *Normal forms* (NF): a set of rules to detect poor database design so they can possibly improved by decompositions.
* Require the concepts of various kinds of *data dependency*: constraints or restrictions between two *sets*of *attributes*.
	1. Functional dependency (FD, most important: used to define NF, up to BCNF)
	2. Multi-valued dependency (MVD for defining 4NF)
	3. Join dependency (JD fordefining 5NF)
* Some common normal forms in ascending order: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF, DKNF, 6NF.
* Higher normal forms are more restrictive and signify better designs.
* A relation in a higher normal form implies that it is in a lower normal form, but not vice versa.

***Example***

If a relation R1 is in 4NF, then R1 is also in BCNF, 3NF, 2NF and 1NF. Refer to the diagram below for R1, R2, and R3.

If a relation R2 is in 2NF, then

1. It is in 1NF,
2. it may or may not be in 3NF, and
3. it may or may not be in BCNF.

If a relation R3 is not in 3NF, then

1. It is not in BCNF.
2. It may or may not be in 1NF or 2NF.

The relations R1 and R3 in this example are depicted in the Venn's Diagram:



**General Overview**

* In general, the higher the normal form a relation is in, the better the logical design of the relation (in terms of avoiding redundancy and inconsistency).
* However, it may be necessary to consider other issues, especially performance.
	1. Higher normal forms may be achieved by *decompositions*, resulting in more relations.
	2. More joins may then be needed to provide the data for a query, decreasing performance.
* 1NF is usually assumed. However, there are relations not in 1NF in both theory and practice.
	1. For example, a composite data type may be supported by a specific DBMS vendor.
* 2NF is more interesting for historical reasons.
* 4NF or above involves data dependency that is hard to understand and use. They are usually not used in practice.
* Based on the concept of functional dependencies (FD), the most important normal forms are
	1. *3NF* and
	2. *BCNF* (Boyce-Codd Normal Form).

**2. Functional Dependencies (FD)**

* Each attribute in a database represents certain data information in the application.
* There can be dependency between data.
* For example, types of dependency and relationship between two *sets* of *attributes* can be:
	+ Many to one (0..\* to 0..1): Functional Dependency (FD)
	+ Many to many (0..\* to 0..\*): Multi-Valued Dependency (MVD)
* These relationships are the results of *assumptions* we made about the application requirements.

***Example:***

A student, {StudentId: 1233457} can be associated with *only one* GPA: {GPA: 3.00}
However, several students can have the same GPA: {GPA: 3.00}

StudentId -> GPA    (FD)
(many)         (one)

On the other hand, a student {StudentId: 1233457} can take*many* courses: {cid: CSCI4333}, {cid:CSCI 2315}, {cid: CSCI3331},...
Many students can take the same course: {StudentId: 1233457}, {StudentId: 2233490}, {StudentId: 3333457},... take {cid: CSCI4333}.

StudentId ->-> cid    (MVD: not covered in this course)
(many)            (many)

**2.1 Many to one relationships**

***Example***

For many applications, the relationship between SSN and FName are many to one in a relation R(..,SSN, FName, ...)

SSN        ->     FName
(many)                 (one)

**Assumptions:**

1. A SSN uniquely identifies a person.
2. Given a SSN, there can only be one first name associated with it (not allowing/storing alias, etc.)
3. Many different SSN's (persons) may have the same first name.
4. There should not be two tuples with the same SSN, but different FName in*all instances* of R.

**Terms:**

1. SSN uniquely determines FName.
2. FName is functionally determined by SSN.
3. There is a functional dependency SSN -> FName.
4. Hence, a functional dependency specifies a many to one relationship between two*sets*of attributes.

For example, the relation instance:

|  |  |  |  |
| --- | --- | --- | --- |
| **SSN** | **FName** | **PHONE** | ... |
| ***123456789*** | ***Peter*** | 123-456-7890 |   |
| ***123456789*** | ***Paul*** | 713-283-7066 |   |
| 222229999 | Mary | 713-283-7066 |   |

is not allowed if we assume SSN -> FName.

***Example***

In a university, there may be a many-to-one relationship between {CourseId, StudentId} and {Grade}.

Interpretations:

1. A student may have only one grade for a course.
2. We say that there is a FD:
	* CourseId, StudentId -> GRADE, or
	* {CourseId, StudentId} determines Grade.
3. Note that under a different set of assumptions, the functional dependency may not be true.
4. For example, if a student is allowed to retake a course, then he may have two grades for the same course (in different semesters), then CourseId, StudentId -> Grade  is false.
5. We may instead have {CourseId, StudentId, Semester, Year}-> {Grade}
* Hence, a functional dependency is a result of the requirements and business logics of the applications.
* There is no universally true non-trivial functional dependency.
* In other words, FD depend on the semantic of the problems.

Note that AB->CD is a shorthand notation for {A,B} -> {C,D}

FD such as AB-> A, AB->B, AB->AB, A->A are *trivial*. They are always mathematically true but do not capture any data requirements.

***Example***:

In most applications, we have

SSN -> FName             (i.e.  a person has only one SSN.)

However, in a criminal database, several bad guys may use the same fake SSN, and thus

SSN -> FName  may not be true.

Or, if you are dealing with an international database with many countries, each country may has its own SSN.  Two countries may issue the same SSN.  Hence,

SSN -> FName   is not true.

We may instead have  SSN, CountryId -> Name.

**2.2 Definition of FD**

* A relation *schema* R is said to satisfy the functional dependency X -> Y if for *any* relation *instance* r that uses R, if there exists two tuples s and t ∈ r such that s[X] = t[X], then s[Y] = t[Y].
	1. (∃s, t ∈ r) (s[X] = t[X]) => s[Y] = t[Y]
	2. i.e.  same value in X implies same value in Y.

***Example***:

This instance r of R violates X->Z.

|  |  |  |
| --- | --- | --- |
| **X** | **Y** | **Z** |
| *'A'* | 1 | *110* |
| *'A'* | 1 | *123* |
| 'A' | 1 | 345 |
| 'B' | 2 | 232 |
| 'C' | 1 | 110 |
| 'C' | 2 | 212 |

This instance r of R *does not violate* X->Y.

|  |  |  |
| --- | --- | --- |
| **X** | **Y** | **Z** |
| 'A' | 1 | 110 |
| 'A' | 1 | 123 |
| 'A' | 1 | 345 |
| 'B' | 2 | 232 |
| 'C' | 1 | 110 |
| 'C' | 1 | 211 |

However, this instance r *does not prove* that X->Y.

In order to have X-> Y, *all* instances r of R must not violate the conditions.

***Examples***:

DeptId -> ManagerId:

There are no two tuples with the same DeptId but different ManagerId.  Meaning: a department can have only one manager.

CourseId, StudentId, Semester -> Grade

There are no two tuples with the same CourseId, StudentId and Semester, but different Grade.  Meaning: any student taking a course in a semester has an unique grade. Note that it may not be true for a different university. Instead, the following may be true:

CourseId, StudentId, Year, Semester -> Grade

***Example***

Consider the following relation:

Supply(SupplierId, SupplierName, ProductId, ProductDesc, Quantity, ArrivalTime)

The relation stores the quantities and arrival times of shipments of products (identified by ProductId) from suppliers (Identified by SupplierId). A supplier may*not* have a unique name. Furthermore, the product description, ProductDesc, may be the same for two products. A supplier may supply the same product many times, each with a different ArrivalTime.

The functional dependencies (FD) of the relation may be:

SupplierId -> SupplierName
ProductId -> ProductDesc
SuplierId, ProductId, ArrivalTime -> Quantity

Decomposition:

Supplier(SupplierId, SupplierName) {SupplierId -> SupplierName}
Product(ProductId, ProductDesc) {ProductId -> ProductDesc}
Supply(SuplierId, ProductId, ArrivalTime, Quantity) {SuplierId, ProductId, ArrivalTime -> Quantity}

***Example (from Spring 2019 HW):***

Consider the following relation GO:

GO(GroupId, GroupName, GroupEMail, GroupChairId, GroupChairLName, GroupChairFName, GroupMemberId, GroupMemberMajor)
The relation stores information about student groups, their chair persons and members. Chair persons and members are students with unique student ids (stored as values in GroupChairId and GroupChairLName respectively). GroupId uniquely identifies a group, and a group has a unique name, and an email address (that may not be unique.) For example, three tuples are shown below.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **GroupId** | **GroupName** | **GroupEMail** | **GroupChairId** | **GroupChairLName** | **GroupChairFName** | **GroupMemberId** | **GroupMemberMajor** |
| G1 | Biology | bio@uhcl.edu | 12345 | Lee | Bryan | 23323 | Biol |
| G1 | Biology | bio@uhcl.edu | 12345 | Lee | Bryan | 24990 | Biol |
| G1 | Biology | bio@uhcl.edu | 12345 | Lee | Bryan | 38879 | Phys |

Bryan Lee is the chair student of the group G1 Biology. The three tuples also store information of three members of group G1

(a) List all applicable functional dependencies. (Make reasonable assumptions if necessary.)

GroupId -> GroupName, GroupEMail, GroupChairId
GroupName -> GroupId
GroupChairId -> GroupChairLName, GroupChairFName
MemberId -> GroupMemberMajor

Q: GroupName -> GroupEMail, GroupChairId? True; inference with FD.

Assumptions:

1. A group has a unique chairperson.
2. A student may be a chairperson or a member for multiple groups.
3. A student has a unique major (e.g., no double majors).

(b) What are the candidate keys?

{GroupId, MemberId} and {GroupName, MemberId}

(c) What is the highest normal form? Why?

1NF. For example, GroupId -> GroupEMail violates 2NF.

(d) If the highest normal form is not BCNF, can you decompose the relation GD losslessly into component relations in BCNF while preserving functional dependencies? If yes, how. If no, why?

1. Group(GroupId, GroupName, GroupEMail, GroupChairId) {GroupChairId references Student(StudentId)}
2. Membership(GroupId, MemberId) {MemberId references Student(StudentId)}
3. Member(StudentId, StudentLName, StudentFName, Major, …)

Note that changes of attribute names in the member tables. For example, StudentLName is more appropriate than ChairLName since a student may not be a chair.

**Theory of Functional Dependency**

by K. Yue

**1. Armstrong's axioms**

* Armstrong's axioms is an inference system.
* Reasoning from the *definition*is difficult. Thus, people invent axioms as an equivalent way for*inference*.
* A relation *schema* R is said to satisfy the functional dependency X -> Y if for *any* relation *instance* r that uses R, if there exists two tuples s and t ∈ r such that s[X] = t[X], then s[Y] = t[Y].
	1. (∃s, t ∈ r) (s[X] = t[X]) => s[Y] = t[Y]
	2. i.e.  same value in X implies same value in Y.
* GroupId (Y) -> GroupName (P), GroupEMail (Q), GroupChairId (R)
GroupName (X) -> GroupId (Y)
GroupChairId -> GroupChairLName, GroupChairFName
MemberId -> GroupMemberMajor
* Q: GroupName –(Z) > GroupEMail (Q), GroupChairId (R)? True; inference with FD.

[1] X -> Y (given)
[2] Y -> PQR (PQR) (given)
[3] X -> PQR (Transitivity axiom on [1] and [2])
[4] X -> QR

* A set of axioms for inference with FD: <http://en.wikipedia.org/wiki/Armstrong%27s_axioms>.
* Axioms: 'self-evidence' or 'assumed' so that they can be used as the basis of inference.
* Three basic axioms:
	1. Reflexivity: If X and Y are sets of attributes and Y is a subset of X, then X -> Y. (non-trivial FD: AB -> A)
	2. Augmentation: If X -> Y then X Z -> Y Z (LHS and RHS are both augmented by Z)
	3. Transitivity**:**If X -> Y and Y -> Z then X -> Z
* Three additional rules that can be derived from the basic axioms.
	1. *Pseudo-transitivity* Rule: If X-> Y, YZ -> A then XZ -> A
	2. Decomposition Rule: If X -> Y Z, then X -> Y and X -> Z. (Note that the decomposition applies to the RHS of the FD)
	3. Union Rule:  If X -> Y and X -> Z then X -> Y Z. (Note that the decomposition applies to the RHS of the FDs)
* Armstrong's axioms are sound and complete.
	1. Soundness: implies only FD that are correct.
	2. Completeness: can be used to derive *all* correct FD.
* Computing students need to know how to infer using a formal mathematical method.

***Example***

Let X be CITY STREET, Y be STREET, then Y is a subset of X, and X -> Y or CITY STREET -> STREET (Reflexivity).

* If two tuples have the same values of CITY and STREET, then they surely have the same value of STREET.
* This is so trivial that we call a functional dependency likes CITY, STREET -> STREET a trivial functional dependency. They do not actually specify any problem requirement.

***Example:***

For R(A,B), we have the following trivial FD for the attributes A and B. No matter what A and B are supposed to mean, they are always mathematically true. (Φ is the empty set.)

AB -> AB, AB->A, AB->B, AB-> Φ (reflexivity axiom: If X and Y are sets of attributes and Y is a subset of X, then X -> Y. (non-trivial FD: AB -> A)
A -> A, A-> Φ
B -> B, B-> Φ

Remember AB-> AB means {A, B} -> {A, B}.

* Since trivial functional dependencies do not actually represent any problem requirements, we are only interested in non-trivial functional dependency. Non-trivial FD are FD in which its RHS is not a subset of its LHS.

If EmpId  ->  DeptId, and DeptId  ->  ManagerId
then EmpId  ->  ManagerId.

Interpretation: If

1. every Employee works for only one department, and
2. every department has only one manager,

then every Employee has only one manager.

A mathematical proof using Armstrong's axiom is to continuously create new FDs until the result is included. Reasons are usually given.

***Example***

Prove that the decomposition rule is true: X->YZ => X->Y and X->Z

Proof:

[1] X->YZ (given)
[2] YZ -> Y (reflexivity axiom)
[3] X -> Y (transitivity axiom on [1] and [2]).
[4] YZ -> Z (reflexivity axiom)
[5] X -> Z (transitivity axiom on [1] and [4]).

**2. Keys and Superkeys Revisited**

* We can use the concept of FD to define keys and superkeys.
* For a relation scheme R, K is a candidate key (CK) if
	1. Uniqueness:  K -> R.
	2. Minimality:  there is no *proper* subset of K that determines R. (There is no extraneous attribute.)
* Note that
	1. |A| = cardinality of A = the number of elements in the set A.
	2. A ⊆ B means A is a subset of B and it is possible that A = B.
	3. A ⊂B means A is a proper subset of B in which A <> B.
	4. If A ⊆ B, |A| <= |B|.
	5. If A ⊂ B, |A| < |B|.
* K is a superkey if K -> R.
* Superkeys (SK) do not need to satisfy the minimality requirement.
* Some properties:
	1. If K is a CK, any superset of K is a SK.
	2. If K is a CK, any proper subset of K is not a CK.
	3. If K is a CK, any proper superset of K is not a CK.
* Note that the primary key of a table is just a selected candidate key used to structure the physical storage. It is just like other candidate keys (*alternate or secondary keys*) in the context of the normalization theory.
* A CK with only one attribute is known as a *simple key*.
* A CK with more than one attributes is known as a *composite key*.
* A *compound* key is a composite key in which every component attribute is a foreign key.

***Example***

In Employee(EmpId, DeptId, ManagerId) with

EmpId -> DeptId and
DeptId -> ManagerId.

By the transitivity axiom, EmpId -> ManagerId
By the union rule, EmpId (K) -> EmpId, DeptId, ManagerId (R)
By the augmentation axiom, EmpId, ManagerId -> DeptId, ManagerId

Hence, EmpId is a CK of Employee(EmpId, DeptId, ManagerId).

On the other hand,

1. DeptId is not a candidate key since we do not have DeptId -> EmpId.
2. {Empd, DeptId} is not a candidate since it is not minimal. It is a superkey only.

Furthermore, there are four superkeys:

1. EmpId
2. EmpId, DeptId
3. EmpId, ManagerId
4. EmpId, DeptId, ManagerId

**3. Finding All Candidate Keys**

**3.1 Closure of Attributes**

* Given a set of FD F, the *closure* of a set of attributes X, denoted as X+, is the set of all attributes functionally determined by X using Armstrong's axioms on F.

***Example***

Consider R(A,B,C,D) with

F = {B->A, A->C, AB->D, D->AC}

A+ = AC (A -> AC but no more)

A+ : [1] A
 [2] A C {A in A+; A-> C}

B+ = ABCD = R => B is a CK.

B+: [1] B
 [2] B A {B in B+; B->B; B-> A => B->A}
 [3] B A D {AB in B+; B-> AB; AB -> D => B -> D by transitivity}
 [4] BAD C {D in D+; B -> D; D->AC => D ->AC by transitivity}
C+ = C
D+ = ACD (D -> ACD; D -x-> B)

Thus, B is a candidate key (CK).

No *proper* superset of B is a candidate key (since it will not be minimal).

Remaining non-empty subsets of ABCD to check for candidate keys:

AC+ = AC
AD+ = ACD
CD+ = ACD
ACD+ = ACD

Thus, B is the only CK.

* The closure of attributes can be used for other purposes, such as checking validity of FD, computing closure of a set of functional dependencies, checking equivalence of two set of FDs, etc.

**3.2 Algorithm for finding X+ for a set of FDs F.**

[1] X+ <- X (X -> X)
[2] while ([A] there exists a FD P -> Q such that [B] P is a subset of X+, and [C] there are attributes K in Q not in X+) { // X -> P; P->Q => X -> Q
   [3] X+ <- X+ U Q      // Add attributes in Q to X+ by using the union operator.
}

**3.3 Finding All Candidate keys**

* It is necessary to find all candidate keys to conduct normalization analysis.
* In general, if R has n attributes, there are 2n - 1 subsets of R which are potential candidate keys.

***Example:***

For R(A,B,C), need to check A, B, C, AB, AC, BC and ABC for candidate keys.

Thus, the problem of finding all candidate keys in R is O(en), where n is the number of attributes in the relation R.

**3.4 To find all candidate keys of R with a set of FD, F:**

1. Additional Material: Find the *canonical cover*, FC, first. This simplifies F. (This step is beneficial but not mandatory. See below.)
2. Use heuristics to cut down the number of sets of attributes to check.
3. Classify attributes into three groups:
	1. L/NR (left only or not right): If an attribute X does not appear in the right hand side (RHS) of any f in F, *every* candidate key must include X.
	2. R (right only): If X appears only in the RHS of a fd in F but does not appear in the LHS of any f in F, then x is *not* a part of *any* candidate key.
	3. M (mixed; left and right): If X appears in LHS in some FD and in RHS in some other FD in F, then X *may* potentially be in some CK.
4. If X is found to be a CK, then any proper superset of X is not a CK, and needs not be checked.

***Example:***

Consider R(A,B,C,D) with

F = {B->A, A->C, AB->D, D->AC}

B: StuId
D: DeptId
A: DeptEmail
C: DeptPhone

We have:

Left/NR (not right: does not appear in the RHS of any FD): B (in every CK)
M: A, D (may be in some CK)
R: C (not in any CK)

Checking: B and then BA, BD, BAD (if needed).

B+: BACD B -> BACD

Thus, there is only one CK: [1] B.

**4. FD Closure and Covers (If time permits)**

**4.1 Closure of a set of functional dependencies**

* The closure of a set of FD, F, is denoted by F+, and is the set of all FDs that are*logically implied* by F.

Consider F = {A->B, B->C}

F+ = {
A->{}, A->A, A->B, A->C, A-> AB, A-> AC, A-> BC, A->ABC,
B->{}, B->B, B->C, B->BC,
C->{}, C->C,
AB->{}, AB->A, AB->B, AB->C, AB->AB, AB->AC, AB->BC, AB->ABC,
AC->{}, AC->A, AC->B, AC->C, AC->AB, AC->AB, AC->BC, AC->ABC,
BC->{}, BC->B, BC->C, BC->BC,
ABC->{}, ABC->A, ABC->B, ABC->C, ABC-> AB, ABC-> AC, ABC-> BC, ABC->ABC }

Note that

* Many FDs in F+ are trivial. Examples: A->{}, ABC->AC, etc.
* FD+ itself is not very interesting.

**4.2 Equivalence and cover**

* Two sets of FD, F and G are equivalent, if F+ = G+. They are covers of each other.
* Thus, covers can be used to support the concepts of equivalence. If F and G are covers of each other, they represent the same set of application requirements and assumptions.

**4.3 Canonical and Minimal Covers**

* **Definition.** In a set of FDs F, the attribute A in the FD P-> Q is extraneous if F - {P-> Q} U {P-A -> Q} is equivalent to F.
* Thus, the attribute A is not actually needed in P to determine Q. It is extraneous.

***Example***

Consider the F = {A->B, AB->C}.

B is extraneous since for G = {A->B, A->C}, and F+ = G+.

* **Definition**. A FD f in F is redundant if (F - f)+ = F+.

***Example***

In F = {A->B, AB->C, B->C},

AB->C is redundant since for

G = {A->B, B->C}, AB+ = ABC.

Alternatively, we may state that

G |- AB-> C.

***Example***

For F = {A->BC, B->C}

Using decomposition rule,

F' = {A->B, A->C, B->C} is a cover of F.

In F', A->C is redundant since {A->B, B->C} |- A->C

Thus F" = {A->B, B->C} is a cover of F' and F.

* **Definition.** A *canonical cover*, G, of F satisfies the following conditions:
	1. G is a cover of F; G+ = F+.
	2. There is no redundant FD in G.
	3. There is no extraneous attribute in G.
	4. The left hand side (LHS)of every FD in G is unique.
* **Definition.** A *minimal cover*, G, of F satisfies the following conditions:
	1. G is a cover of F; G+ = F+.
	2. There is no redundant FD in G.
	3. There is no extraneous attribute in G.
	4. The right hand side (RHS) of every FD in G contains only a single attribute

In F = {A->B, AB->C, B->C, A->D},

G1 = {A->B, B->C, A->D} is a minimal cover.

G2 = {A->BD, B->C} is a canonical cover.

* The minimal covers and canonical covers are simplified equivalent versions of a set of FDs, representing the same set of data requirements.
* They are useful in understanding FD and for proper decompositions to remove unnecessary redundancy.

***Example:***

Consider F: {A->C, BCD->A, C->E, CD-> A, AB->C}

[1] Does F imply BD-> A (i.e. F |- BD -> A)?

No, Since in F, BD+ = BD

THus, C is not extraneous in BCD -> A.

[2] F |- AE -> B ?

No, since AE+ = AE C

[3] Give a canonical cover for F.

{ A->C, CD->A, C->E }

[4] Show all candidate keys.

L/NR: B, D
M: A, C
R: E

BD+: BD

CK: [1] ABD, [2] CBD

Prime attributes: A, B, C, D

Non-prime attributes: E

***Example (Tedious):***

Find a canonical cover for F = {BC->AE, AD->BCE, A->E, AE->D, BCD->F, AB->C}

***Solution:***

Basically, we iteratively remove all extraneous attributes and redundant function dependencies.

We use decomposition rule to ensure the RHS to contain only a single attribute so we can work on them one by one. F becomes:

(1) BC -> A
(2) BC -> E
(3) AD -> B
(4) AD -> C
(5) AD -> E
(6) A -> E
(7) AE -> D
(8) BCD -> F
(9) AB -> C

To investigate whether B or C is extraneous in BC -> A, we note that in F:

B+ = B
C+ = C

This means B alone and C alone cannot determine A, and neither of them is extraneous.

On the other hand, in F:

A+ = ABCDEF

That means A alone can determine all other attributes. Any other attributes in the LHS with A in a FD are thus extraneous, we thus have the following by removing D in [2], [3] and [4], and B in [9].

(1) BC -> A
(2) BC -> E
(3) A -> B
(4) A -> C
(5) A -> E
(6) A -> E
(7) A -> D
(8) BCD -> F
(9) A -> C

Removing identical FD, we have F:

(1) BC -> A
(2) BC -> E
(3) A -> B
(4) A -> C
(5) A -> E
(6) A -> D
(7) BCD -> F

For (7), since B+ = B, C+ = C and D+ = D. However, BC+ = ABCDEF, and thus D is extraneous. Thus, we now have:

(1) BC -> A
(2) BC -> E
(3) A -> B
(4) A -> C
(5) A -> E
(6) A -> D
(7) BC -> F

To check for redundant FD, we consider whether we can deduce the FD when it is removed.

For (1) BC -> A, removing it result in F':

(1) BC -> E
(2) A -> B
(3) A -> C
(4) A -> E
(5) A -> D
(6) BC -> F

In F': we have

BC+ = BCE, which does not include A. Thus, F' does not imply BC -> A and it is not redundant.

For (2) BC -> E, removing it and we have F':

(1) BC -> A
(2) A -> B
(3) A -> C
(4) A -> E
(5) A -> D
(6) BC -> F

In F', we have BC+ = ABCDEF. Thus, F' |= BC -> E and BC -> E is redundant. Remove it and we have:

(1) BC -> A
(2) A -> B
(3) A -> C
(4) A -> E
(5) A -> D
(6) BC -> F

Using this method, we can find that there are no more redundant FD.

Finally, we use the union rule to merge FD with the same LHS and get the canonical cover:

{BC -> AF, A-> BCDE}

Note that the canonical cover is not unique. Another canonical cover is:

{BC -> A, A-> BCDEF}

***Exercise:***

Consider F: {AB->CE, BC->D, D->BC, C->E, A->C, A->E}

Find:

* all candidate keys.
* a canonical cover of F.

***Exercise:***

Can there be more than one canonical covers for a set of FDs?