# CSCI 5333 DBMS Classroom Notes

**11/23/2021**

**Normalization Theory**

by K. Yue

**1. Functional Dependencies**

* Normal forms: a set of rules to avoid redundancy and inconsistency.
* Require the concepts of data dependencies. Examples:
  1. functional dependency (FD, most important: up to BCNF)
  2. multivalued dependency (MVD for 4NF)
  3. join dependency (5NF)
* Common Normal Forms in ascending order: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF, DKNF, 6NF.
* Higher normal forms are more restrictive.
* A relation is in a higher normal form implies that it is in a lower normal form, but not vice versa.

***Example:***

If a relation R is in BCNF, then R is also in 3NF, 2NF and 1NF.

If a relation 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 relations is not in 3NF, then

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

* In general, the higher the normal forms a relation is in, the better the design of the relation in terms of avoiding redundancy and inconsistency is.
* However, it may be necessary to consider other issues, especially performance.
  + Higher normal forms may be achieved by decomposition, resulting in more relations. 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.
  + For an example, a composite data type may be supported by a specific DBMS vendor.
  + Standard SQL supports many non-1NF features.
* 2NF are more interesting for historical reasons.
* 4NF and 5NF involves the concept of multivalued and join dependencies (MVD and JD). They are hard to understand and even harder to apply in most situations.
* Domain Key Normal Form (DKNF) involves the concept of constraints.
* Based on the concept of functional dependencies (FD), the most important normal forms are
  + 3NF and
  + BCNF (Boyce-Codd Normal Form).

**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*:
  + Many to one (0..\* to 0..1): FD
  + Many to many (0..\* to 0..\*): MVD
* These relationships are the results of *assumptions* we made about the application requirements.

***Example***

Many to many relationships.

Consider an instance of the relation Enroll:

|  |  |  |
| --- | --- | --- |
| **Course** | **Student** | **Grade** |
| C1 | S1 | A |
| C1 | S2 | B |
| C1 | S3 | B |
| C2 | S1 | A |
| C2 | S4 | D |

Under reasonable assumptions, there are many to many relationships between these sets of attributes:

1. Course and Student: A course may enroll many students. A student may take many courses.
2. Course and Grade
3. Student and Grade
4. {Course, Grade} and Student: both S2 and S3 may have a grade of B in Course C1.

However, the relationship between {Course, Student} and Grade may not be a many-to-many relationship if we assume that a student can only has one grade for a given course.

* A many to many relationship between two sets of attributes means that there is no functional dependency between the values of these two sets of attributes.

***Example***

Many to one relationships.

For many applications, the relationship between the two sets of attributes, SSNUM and NAME, are many to one.

SSNUM        ->     NAME  
(many)                (one)

**Assumptions:**

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

**Terms:**

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

For example, the relation instance:

|  |  |  |
| --- | --- | --- |
| **SSNum** | **NAME** | **PHONE** |
| **123456789** | **Peter** | 123-456-7890 |
| **123456789** | **Paul** | 713-283-7066 |
| 222229999 | Mary | 713-283-7066 |

is not allowed if we assume SSNum -> NAME.

***Example***

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

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 functional dependency:
   * CourseId, StudentId -> GRADE, or
   * {CourseId, StudentId} determines Grade.
3. Note that under different 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 actually have CourseId, StudentId, Semester -> Grade

* Hence, 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, functional dependencies depend on the semantic of the problems.

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

***Example***

In most applications, we have

SSNum -> Name             (i.e.  a person has only one SSNum.)

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

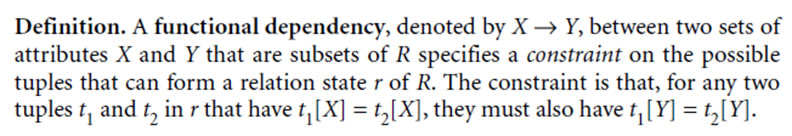
SSNum -> Name  is not true.

Or, if you are dealing with an international data base with many countries.  Each country may has its own SSNum.  Two countries may issue the same SSNum.  Hence,

SSNum -> Name   is not true.

We may instead have  SSNum, Country -> Name.

**Definition of FD** (from EN):



A relation scheme R is said to satisfy the functional dependency X -> Y if for any relation instance r that uses R (relation schema), if there are two tuples s and t in r such that s[X] = t[X], then s[Y] = t[Y].

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

Does Z -> X? No

Does Z -> Y? No violation in this instance. Maybe it is true.

|  |  |  |
| --- | --- | --- |
| **X** | **Y** | **Z** |
| 'A' | 1 | 110 |
| 'A' | 1 | 123 |
| 'A' | 1 | 345 |
| 'B' | 2 | 232 |
| 'C' | 1 | 110 |
| ‘C’ | 3 | 130 -> violate X->Y |

This instance 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 |

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

***Example***

SSNUM -> SNAME:

There are no two tuples with the same SSNUM but different names.

DEPT\_NO -> MANAGER\_NO:

There are no two tuples with the same DEPT\_NO but different MANAGER\_NO.  A department has only one manager.

SNUM, PNUM, DATE -> QUANTITY

There are no two tuples with the same SNUM, PNUM and DATE but different QUANTITY.  That is, any supplier has only one shipment of a part on a given date.

In the example of poorly designed database:

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_NO** | **NAME** | **DEPT\_NO** | **MANAGER\_NO** |
| 10000 | Lady Gaga | D123 | 54321 |
| 12000 | Aamir Khan | D123 | 54321 |
| 13000 | Lebron James | D123 | 54321 |
| 21000 | Narendra Modi | D225 | 42315 |
| 22000 | Aishwarya Rai | D225 | 42315 |
| 31000 | John Smithson | D337 | 33323 |

If we assume that a department has only one manager, we have:

DEPT\_NO -> MANAGER\_NO

Note that we also have:

NAME, DEPT\_NO -> MANAGER\_NO  
EMP\_NO, DEPT\_NO -> MANAGER\_NO

and so on.

# A FD Exercise:

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: may be shared by many organizations?.) 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 |
| G1 |  | biology@gmail.com? Ask. |  |  |  |  |  |
|  |  |  |  |  |  | 38879 | CS? |

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

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

GroupName -> GroupId? Yes

1. Yes => Group name can determine/identify a group.
2. No => a group may have multiple group names.

GroupMemberId -> GroupMemberMajor? Yes

1. Yes -> a member has only one major.
2. No => a member may have double majors.

GroupChairId -> GroupChairLName, GroupChairFName

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

GroupId -> GroupMemberId, GroupMemberMajor? No, because a group has many members.

GroupId -> GroupChairName? Yes, a group has one chairperson with one name.

GroupId -> GroupChairId? Yes, a group has one chairperson.

GroupId -> GroupName? Yes

GroupId -> GroupEMail? Yes, assume that a group has only one email address on the record.

There are trivial FDs that are always true. Yes, but it does not represent any requirements. E.g. A->A, AB->A, A-> {}

(b) What are the candidate keys?

R = GroupId, GroupName, GroupEMail, GroupChairId, GroupChairLName, GroupChairFName, GroupMemberId, GroupMemberMajor

GroupMemberId -> GroupMemberMajor

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

{GroupId, GroupMemberId} -> GroupId, GroupMemberId, GroupName, GroupEMail, GroupChairId, GroupChairFName, GroupChairLName, GroupMemberMajor = R

GroupName -> GroupId

{GroupName, GroupMemberId} -> GroupId, GroupMemberId, GroupName, GroupEMail, GroupChairId, GroupChairFName, GroupChairLName, GroupMemberMajor = R

CK: (1) {GroupId, GroupMemberId}; (2) {GroupName, GroupMemberId}

Prime/Key attributes (appear in a CK): GroupId, GroupMemberId, GroupName

Non-prime/non-key attributes (not appear in any CK): GroupEMail, GroupChairId, GroupChairFName, GroupChairLName, GroupMemberMajor

Superkey (SK: unique): any supersets of a CK is a SK (minimality is not required).

Spring 2020 HW #7:

(5) It is known that R(A,B,C,D,E) has exactly two candidate keys. Furthermore, one of the candidate key is known to be AB. What are the maximum and minimum number of superkeys R may have?

AB is a CK (unique + minimal) =>

1. SK: AB, ABC, ABD, ABC, ABCD, ABCE, ABDE, ABCDE: 8 = 23 = number of supersets of AB = {A, B}
2. Not a CK => A, B (not unique), ABC, ABD, ABC, ABCD, ABCE, ABDE, ABCDE (not minimal).

Second CK:

1. C: additional SK (contains C but not AB): C, AC, BC, DC, EC, ACD, ACE, ACDE, BCD, BCE, CDE, BCDE: 12; Total: 20: maximum
2. CD: additional SK (contains CD but no AB): CD, CDE, CDA, CDB, CDAE, CDBE: 6; total: 14.
3. AC:
4. CDE:
5. ACD: additional SK: ACD, ACDE: 2; total: 10
6. ACDE: additional SK: ACDE: 1: Total: 9: minimum

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

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

A relation scheme R is said to satisfy the functional dependency X -> Y if for any relation instance r that uses R (relation schema), if there are two tuples s and t in r such that s[X] = t[X], then s[Y] = t[Y]. Precise and accurate. Not easy to infer/reason. (e.g. A->B, B->C => A->C? Yes)

**Armstrong's axioms**

* A set of axioms for inference with FD: <http://en.wikipedia.org/wiki/Armstrong%27s_axioms>.
* Axioms: 'self-evidence', 'assumed', or 'established'.
* Three basic axioms:
  1. Reflexivity: If X and Y are sets of attributes and Y is a subset of X, then X -> Y.
  2. Augmentation: If X -> Y then X Z -> Y Z.
  3. Transitivity**:**If X -> Y and Y -> Z then X -> Z
* Three additional rules that can be proven by the basic axioms.
  1. *Pseudo-transivitiy* Rule: If X-> Y, YZ -> A then XZ -> A
  2. Decomposition Rule: If X -> Y Z, then X -> Y and X -> Z.
  3. Union Rule:  If X -> Y and X -> Z then X -> Y Z.
* Armstrong's axioms are sound and complete.
  1. Sound: implies only FD that are correct.
  2. Complete: can be used to imply all correct FD.
* CS 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 value 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 business requirement.

A -> A and BC -> B are trivial.

* Since trivial functional dependencies do not actually give you any information, we are only interested in non-trivial*functional dependency*.

If EMP\_NO  ->  DEPT\_NO, and DEPT\_NO  ->  MANAGER\_NO  
then EMP\_NO  ->  MANAGER\_NO

Interpretation: If

* every employee works for only one department, and
* every department has only one manager,

then every employee has only one manager.

**Proof with Armstrong axioms**.

1. Show new facts (FDs) and provide the *reasons*. Each new fact should be numbered for easy reference.
2. Stop when the new fact is the FD to be proved.

***Example***

Prove the union Rule. 1. If X -> Y and X -> Z then X -> Y Z.

**Proof.**

(1) X -> Z (given)  
(2) X X = {X, X} -> X Z (augmentation of (1) with X)  
(3) X -> XZ (simplification of (2))  
(4) X -> Y (given)  
(5) XZ -> YZ (augmentation of (4) with Z)  
(6) X -> YZ (transitivity on (3) and (5))

***Exercise***

Prove the pseudo-transitivity rule.

***Examples***

(1) Prove that F = {AC->B, B->D, AE->C}

implies AE->D

Proof. For example:

[1] AE-> C (given)  
[2] AC -> B (given)  
[3] A:Z AE: X -> B: P (pseudo-transitivity on (1) and (2)): Pseudo-transivitiy Rule: If X: AE -> Y: C, YZ: A -> P: B then XZ -> P  
[4] AE -> B (simplification of (3))  
[5] B -> D (given)  
[6} AE ->D (transitivity on [4] and [5])

(2) Prove that F = {AB->C, AC->D, BD->E}

implies AB->E

Proof. For example:

[1] AB-> C (given)  
[2] AC -> D (given)  
[3] AAB -> D (pseudo-transitivity on (1) and (2))   
[4] AB -> D (simplification of (3))  
[5] BD -> E (given)  
[6] ABB-> E (pseudo-transitivity on (4) and (5))  
[7] AB-> E (simplification of (6))

**Keys and Superkeys**

* We can use functional dependencies 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.)
* 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 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*.

***Example***

In EMPLOYEE(EMP\_NO, DEPT\_NO, MANAGER\_NO) with

EMP\_NO -> DEPT\_NO and  
DEPT\_NO -> MANAGER\_NO.

By the transitivity axiom, EMP\_NO -> MANAGER\_NO.  
By the union rule, EMP\_NO -> EMP\_NO DEPT\_NO, MANAGER\_NO.  
                         
Hence, EMP\_NO is a candidate key of EMPLOYEE(EMP\_NO, DEPT\_NO, MANAGER\_NO).

On the other hand, DEPT\_NO is not a candidate key since we do not have DEPT\_NO -> EMP\_NO.

Furthermore, there are four superkeys:

1. EMP\_NO
2. EMP\_NO, DEPT\_NO
3. EMP\_NO, MANAGER\_NO
4. EMP\_NO, DEPT\_NO, MANAGER\_NO

**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  
B+ = ABCD  
C+ = C  
D+ = ACD

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 subset of ABCD to check:

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

Thus, B is the only CK.

***Example*** (A more degenerate case)

Consider:

F = {A-> B, BC -> DA, BD -> C, E-> A, AC -> DE}

We have

AC+: AC  
 : ACB (A->B)  
 : ACBD (BC -> DA)  
 : ACBDE = R(AC->DE)

AC is a CK.

A+ : A  
 : AB (P->Q: A->B)

E+ : E  
 : EA (E -> A)  
 : EAB (A -> B)

A+ = AB  
B+ = B  
C+ = C  
D+ = D  
E+ = EAB  
(AB)+ = AB  
(AC)+ = ABCDE  
(AD+ = ABCDE  
(AE)+ = ABE  
(BC)+ = ABCDE  
(BD)+ = ABCDE  
(BE)+ = ABE  
(CD)+ = CD  
(CE)+ = ABCDE  
(DE)+ = ABCDE  
(ABC)+ = ABCDE  
...  
(ABE)+ = ABE  
...

* There are thus six candidate keys: AC, AD, BC, BD, CE and DE. Also, all attributes are prime. No non-prime attributes => in 3NF
* This is a theoretical example not likely to appear in the real world, especially if you have performed a good data modeling job.
* 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.

**Algorithm for finding X+ for a set of FDs F. (X -> X+)**

X+ <- X (because X->X)  
while (there exists a FD P -> Q such that P is a subset of X+ and there are attributes K in Q that is not in X+) { // X-> X+ -> P -(reflexivity -> Q, transitivity)   
   X+ <- X+ U Q  
}

***Examples:***

(1) Relation R(A,B,C,D,E) has exactly four superkeys. Can you deduce from this statement the number of candidate keys? If yes, how many CKs are there? Justify your answer.

Solution:

No. If ABC is the only CK of R, then there are four superkeys: ABC, ABCD, ABCE and ABCDE. On the other hand, if there are three CKs: ABCD, ABCE and ABDE, there are also four superkeys: ABCD, ABCE, ABDE and ABCDE.

(2) A relation R of four attributes has two candidate keys, what are the maximum and minimum numbers of superkeys R may have?

Solution:

Minimum: 3; e.g. when the candidate keys are ABC and ABD.  
Maximum: 12; e.g. when the candidate keys are A and B.

(3) Consider the following valid instance of a relation R(A,B,C). Can you deduct from it all candidate keys of R? If yes, what are the candidate keys? If not, why?

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |
| a1 | b1 | c1 |
| a2 | b2 | c1 |
| a2 | b2 | c2 |
| a3 | b1 | c1 |
| a3 | b3 | c1 |

Solution:

Yes, there is only one candidate key: ABC. This is because for all proper subsets of ABC, there are two or more tuples with the same values and thus no proper subsets of ABC can be a candidate key. This leaves only ABC.

**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->{}, 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->AC, 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.

**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.

E.g. F={A->B, B->C}; G = {A->B, B->C, A->C} F+ = G+ G simplifies to F.A->C is redundant in G.

**Canonical and Minimal Covers**

* The attribute A in the FD P-> Q is extraneous for a set of FDs F 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} equivalent to {A->B, A->C}: minimal cover -> {A->BC}: canonical cover of F

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

F = {A->B, AB->C}: B+: B; A+: ABC;

* 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,

G |- AB-> C.

* 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 of every FD in G is unique.
* 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 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,
* They are useful in understanding the FD and for proper decompositions to remove unnecessary redundancy.

***Exercise:***

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

1. Does F imply BD-> A (i.e. F |- BD -> A)?
2. F |- AE -> B ?
3. Give a canonical cover for F.
4. Show all candidate keys.

***Example:***

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: D is extraneous because A+: ABCDEF => A -> B  
(4) AD -> C: D is extraneous because A+: ABCDEF => A -> C  
(5) AD -> E: D is extraneous because A+: ABCDEF => A -> E  
(6) A -> E  
(7) AE -> D: E is extraneous: ABCDEF => A -> D  
(8) BCD -> F: D is extraneous: BC+: BCAEDF => BC -> F  
(9) AB -> C: B is extraneous because A+: ABCDEF => A -> 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 remove redundant FD, we consider whether we can deduce a 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 the minimal cover:

(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?

**Finding Candidate keys**

* It is necessary to find all candidate keys to conduct normalization analysis.
* In general, if R has n attributes, there are 2^n-1 non-empty 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 is O(eN).

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

1. Find the *canonical cover*, FC, first. This simplifies F.
2. Use heuristics to cut down the number of sets of attributes to check.
3. Use classification of attributes into three groups
   1. L (NR): If X does not appear in the RHS of any f in FC, every candidate key must include X.
   2. R: If X appears in the RHS of a fd in FC but does not appear in the LHS of any f in FC, then x is not a part of any candidate key.
   3. M: If X appears in LHS in some FD and in RHS in some other FD, then X can potentially be in a 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:***

For R(A,B,C,D,E,F) with F = {BC->AE, AD->BC, A->E, AE->D, BCD->F, AB->C}, find all candidate keys.

{BC->AE, AD->BC , A->E, AE->D, BCD->F, AB->C} => {BC -> AF, A-> BCDE}

Canonical cover: {BC -> AF, A-> BCDE}

R: DEF (not in any CK)  
L (NR): empty set  
M: ABC

Check A+ => a CK;

Check B+, C+, BC+ => BC is a CK

Solution:

There are 63 potential candidates for the CK. These are the non-empty subsets of ABCDEF.

In F, we have:

A+ = ABCDEF  
B+ = B  
C+ = C  
D+ = D  
E+ = E  
F+ = F

Thus, A is a CK. As a result, it is not necessary to check all proper supersets of A as they will not be minimal.

Continue to check sets of two attributes, not including A:

BC+ = ABCDEF  
BD+ = BD  
BE+ = BE  
BF+ = BF  
CD+ = CD  
CE+ = CE  
CF+ = CF  
DE+ = DE  
DF+ = DF  
EF+ = EF

Thus, BC is another CK.

Further checking on sets of three attributes not including A or BC

BDE+ = BDE  
BDF+ = BDF  
BEF+ = BEF  
CDE+ = CDE  
CDF+ = CDF  
CEF+ = CEF  
DEF+ = DEF

Note that it is not necessary to check proper subsets of A or BC.

We check sets of four attributes:

BDEF+ = BDEF  
CDEF+ = CDEF

We have exhausted all candidates and there are only two CKs: A and BC.

Alternatively, we can use the canonical cover:

{BC -> A, A-> BCDEF}  
  
It can be seen that A and BC are CK. Furthermore, DEF appears in RHS of some FD but not the LHS of any FD, thus any of them can not appear in any CK.

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

F = {A->BCE, BC->D, AB->E, DE->C, AE->CD}

(a) What are A+, B+, C+, D+ and E+?

A+: ABCED = R => A is a CK.  
B+: B  
C+: C  
D+: D  
E+: E

(b) What are the candidate keys? Why?

[1] A

{A->BDE, BC->D, DE->C}

L: A (in every CK)  
M: BCDE (may appear in a CK)  
R: empty

A+: ABCDE =? CK: (1)

(c) Show all prime attributes and non-prime attributes?

Prime: A; Non-prime: BCDE

(d) Give a canonical cover of F?

Extraneous attributes: Check: BC->D, AB->E, DE->C, AE->C, AE ->D

For BC->D: no extraneous because B+=B, C+=C.  
For AB->E: B is extraneous because A+: ABCED => A -> E  
For DE -> C: no extraneous because E+: E, D+: D  
For AE->C: E is extraneous because A+: ABCED => A -> C  
For AE->D: E is extraneous because A+: ABCED => A -> D  
  
F = {A->BCE, BC->D, AB->E, DE->C, AE->CD}

=> F’ = {A->BCE, BC->D, A->E, DE->C, A->CD}

Check redundant FD: {A->B, A->C, A->E, BC->D, ~~A->E~~, DE->C, ~~A->C~~, A->D}

Check A-> B redundant: F2 = {A->C, A->E, BC->D, A->E, DE->C, A->C, A->D} => A->B

For F2: A+: ACED =x=> A->B. Therefore A->B is not redundant.

Check A->C redundant F3 = {A->B, ~~A->C~~, A->E, BC->D, DE->C, A->D} => A->C

In F3, A+: ABEDC. Therefore, A->C is redundant.

F3 = {A->B, A->E, BC->D, DE->C, A->D} is still a cover.

…

Check A->D for redundancy: F4 = {A->B, A->E, BC->D, DE->C, ~~A->D~~} => A->D

IN F4, A+: ABE, A->D is not redundant.

{A->B, A->E, BC->D, DE->C, A->D} -> {A->BDE, BC->D, DE->C}

(e) What is the highest normal form (up to BCNF) of R? Why?

(f) If R is not in BCNF, can you provide a lossless FD preserving decompositions of R into BCNF relations?

**2. Normal Forms using Functional Dependencies**

**First Normal Form**

* A relation is in 1NF if all attribute values are atomic: no repeating group, no composite attributes. (scalar data type)
* Formally, a relation may only has atomic attributes.  Thus, all relations satisfy 1NF.
* In practice, DBMS may allow data types with composite values, e.g. set, list, etc.

Consider the following table with 3 records.  It is not in 1 NF.

|  |  |  |  |
| --- | --- | --- | --- |
| **DEPT\_NO** | **MANAGER\_NO** | **EMP\_NO** | **NAME** |
| D123 | 110 | 10000, 12000, 13000 | Lady Gaga, Eminem, Lebron James |
| D225 | 42315 | 21000, 22000 | Rajiv Gandhi, Bill Clinton |
| D337 | 33323 | 31000 | John Smithson |

The corresponding relation with 6 tuples is in 1 NF:

|  |  |  |  |
| --- | --- | --- | --- |
| **DEPT\_NO** | **MANAGER\_NO** | **EMP\_NO** | **NAME** |
| D123 | 110 | 10000 | Lady Gaga |
| D123 | 110 | 12000 | Eminem |
| D123 | 110 | 13000 | Lebron James |
| D225 | 42315 | 21000 | Rajiv Gandhi |
| D225 | 42315 | 22000 | Bill Clinton |
| D337 | 33323 | 31000 | John Smithson |

* Why atomic? relational theory and operations treat attributes as atomic.
* Relations satisfying only 1NF has unnecessary redundancy and anomalies.

***Example***

Consider the tuple (Empid: 12345, OSSkills: "Windows, Linux, Solaris").

* It will be difficult to identify all employees with Linux skills.
* OSSkills will not be effective as the common attribute of joins.
* Data entry problems and issues, e.g. Linux linux, linx, etc., may further degrade data quality and introduce inconsistency.

**Second Normal Form**

* A relation R is in 2NF if
  1. R is in 1NF, and
  2. all non-prime attributes are fully dependent on the candidate keys. (Any non-prime attribute is not determined by a part of any candidate key.)
* A prime attribute appears in one or more candidate key. Otherwise, it is a non-prime attribute.
* Note that a relation may have many candidate keys.
* A non-prime attribute does not appear in any candidate key.
* There is no partial dependency in 2NF.
* 2NF: If X -> A, A is a non-prime attribute, and X is a subset of a candidate key K, then X = K.

***Example***

The following relation is not in 2NF.  (Assume the number of credits of a given course does not change).  Note the redundancy and anomalies.

Enroll(Course, Credit, Student, Grade): Highest NF: 1NF.

|  |  |  |  |
| --- | --- | --- | --- |
| **Course** | **Credit** | **Student** | **Grade** |
| C1 | 3 | S1 | A |
| C1 | 3 | S2 | B |
| C1 | 3 | S3 | B |
| C2 | 2 | S1 | A |
| C2 | 2 | S4 | D |

That is, we assume the following FDs.

1. Course (a proper subset of a CK)-> Credit (non-prime): violates 2NF.
2. Course, Student (full CK) -> Grade (non-prime): ok with 2NF

L/NR: Course, Student (every CK)  
R: Credit, Grade (not in any CK)

Thus,

1. {Course, Student} is the only candidate key.
2. Prime attributes: Course, Student
3. Non-prime attributes: Credit, Grade.
4. FD (1) is a violation of 2NF.

To convert to 2NF, decompose Enroll into

1. Enroll(Course, Student, Grade): {Course, Student (full CK) -> Grade (non-prime)}: ok with 2NF
2. Class(Course, Credit) {Course (full CK)-> Credit (non-prime)}: ok with 2NF

|  |  |
| --- | --- |
| **Normal Form** | **Violating Non-trivial FD X-> Y** |
| 2NF | [1] X is a proper subset of a CK, and [2] Y is non-prime |
| 3NF | [1] X is not a SK, and [2] Y is non-prime |
|  |  |

**Third Normal Form**

* (Old definition) A relation R is in 3NF if
  1. R is in 2NF, and
  2. There is no transitive dependency of nonprime attributes on the keys.
* (New definition) : a relation R is said to be in the third normal form if for every non-trivial functional dependency X -> A,
  1. P: X is a superkey, or
  2. Q: A is a prime (key) attribute.
* Violating FD: -(P V Q) = -P and -Q.

***Example***

* The following relation may be in 2NF, but is not in 3NF.

|  |  |  |  |
| --- | --- | --- | --- |
| **DEPT\_NO** | **MANAGER\_NO** | **EMP\_NO** | **NAME** |
| D123 | 54321 | 10000 | Lady Gaga |
| D123 | 54321 | 12000 | Eminem |
| D123 | 54321 | 13000 | Lebron James |
| D225 | 42315 | 21000 | Rajiv Gandhi |
| D225 | 42315 | 22000 | Bill Clinton |
| D337 | 33323 | 31000 | John Smithson |

* If we assume the following canonical set of FDs:
  1. EMP\_NO (full CK; SK) -> NAME, DEPT\_NO (non-prime): ok with 2NF, 3NF
  2. DEPT\_ NO (not a part of a CK; not a SK) -> MANAGER\_NO (non-prime): ok with 2NF; not ok with 3NF
* EMP\_NO, NAME (a SK, not a CK) -> MANAGER\_NO (not violate 3NF)
* then
  1. There is only one candidate key: EMP\_NO
  2. Prime attributes: EMP\_NO
  3. Non-prime attributes: NAME, DEPT\_NO, MANAGER\_NO.
  4. The relation is in 2NF.
  5. The relation is not in 3NF because of the transitive FD: EMP\_NO -> MANAGER\_NO via the non-prime attribute DEPT\_NO.

Using the new definition.

The functional dependency  DEPT\_NO -> MANAGER\_NO is

(1) non-trivial,  
(2) DEPT\_NO is not a superkey, and  
(3) MANAGER\_NO is not a prime attribute.

Thus, it violates the 3NF.

Decomposition:

DEPT(DEPT\_NO, MANAGER\_NO) { DEPT\_ NO (a CK; a SK) -> MANAGER\_NO (non-prime)}  
EMP(EMP\_NO, NAME, DEPT\_NO) { EMP\_NO (full CK; SK) -> NAME, DEPT\_NO}

Scope and context:

***Example***

For the relation R(CITY, ZIP, STREET)  
         
Using the code for the postal office, we may have:

CITY STREET -> ZIP, and ZIP -> CITY.

Hence, there are two candidate keys:

1. CITY STREET, and
2. ZIP STREET

Hence,

Prime attributes: STREET, CITY, ZIP

R is in the 3NF because

* For the non-trivial FD: CITY STREET -> ZIP, CITY STREET is a superkey.
* For the non-trivial FD: ZIP -> CITY, CITY is a prime attribute.

Note that a relation such as EMPLOYEE(EMP\_ID, EMP\_NAME, Street, City, Zip, State) is not in 3NF.

This is a classical example you can find in many database textbooks. The FDs are actually not valid in the United States. See, for example: [Why all 5-digit ZIP Code™ lists are obsolete](http://maf.directory/zp4/zip5.html).

Note:

* 3NF cannot eliminate all redundancy due to functional dependencies.

***Example***

Consider the relation

S(SNUM, PNUM, SNAME, QUANTITY) with the following assumptions:

1. SNUM is unique for every supplier.
2. SNAME is unique for every supplier.
3. QUANTITY is the accumulated quantities of a part supplied by a supplier.
4. A supplier can supply more than one part.
5. A part can be supplied by more than one supplier.

We have the following non-trivial functional dependencies:

1. SNUM (not a SK) -> SNAME (prime): ok with 3NF; not ok with BCNF
2. SNAME -> SNUM
3. SNUM PNUM (a SK) -> QUANTITY (non-prime): ok with 3NF
4. SNAME PNUM -> QUANTITY

Note that SNUM and SNAME are equivalent.

The candidate keys are:

1. SNUM PNUM
2. SNAME PNUM

Prime attributes: SNUM, PNUM, SNAME

Non-prime attribute: QUANTITY.

S is in 3NF because

* For the non-trivial FDs (1) and (2), the right hand sides are prime attributes (SNAME and SNUM).
* For the functional dependencies (3) and (4), the left hand sides are superkeys.

The relation is in 3NF. However, there are unnecessary redundancy.

|  |  |  |  |
| --- | --- | --- | --- |
| **SNUM** | **SNAME** | **PNUM** | **QUANTITY** |
| *S1* | *ABC* | P1 | 10 |
| *S1* | *ABC* | P2 | 20 |
| *S1* | *ABC* | P3 | 21 |
| S2 | DEF | P1 | 40 |
| S2 | DEF | P4 | 13 |
| S3 | XYK | P3 | 18 |

* 3NF does not eliminate all redundancy due to functional dependencies.

**BCNF (Boyce-Codd Normal Form)**

* A relation R is said to be in **BCNF** if for every non-trivial functional dependency X -> A in R, X is a superkey.

|  |  |
| --- | --- |
| **Normal Form** | **Violating Non-trivial FD X-> Y** |
| 2NF | [1] X is a proper subset of a CK, and [2] Y is non-prime |
| 3NF | [1] X is not a SK, and [2] Y is non-prime |
| BCNF | X is a SK |

***Example***

EMPLOYEE(EMP\_NO, NAME, DEPT\_NO, MANAGER\_NO) with

EMP\_NO -> NAME  
EMP\_NO -> DEPT\_NO  
DEPT\_NO -> MANAGER\_NO

is not in BCNF.

The functional dependency DEPT\_NO -> MANAGER\_NO is

(1)  non-trivial, and  
(2)  DEPT\_NO is not a superkey.

* Recall that this is the example we used for illustrating bad design.
* This is also not in 3NF.

We can decompose

EMPLOYEE(EMP\_NO, NAME, DEPT\_NO, MANAGER\_NO) into

EMP(EMP\_NO, NAME, DEPT\_NO) with

EMP\_NO -> NAME, DEPT\_NO

and

DEPARTMENT(DEPT\_NO, MANAGER\_NO) with

DEPT\_NO -> MANAGER\_NO

Both relations are in BCNF since

* EMP\_NO is a superkey of the relation EMP.
* DEPT\_NO is a superkey of the relation DEPARTMENT.

Recall that these are the good relations without the anomalies in the previous example.

***Example***

Consider again the relation

S(SNUM, PNUM, SNAME, QUANTITY) with the following non-trivial functional dependencies:

1. SNUM -> SNAME
2. SNAME -> SNUM
3. SNUM PNUM -> QUANTITY
4. SNAME PNUM -> QUANTITY

Note that SNUM and SNAME are equivalent.

The candidate keys are:

1. SNUM PNUM
2. SNAME PNUM

Prime attributes: SNUM, PNUM, SNAME

Non-prime attribute: QUANTITY.

S is not in BCNF because, for example, the functional dependency

SNUM -> SNAME is

* non-trivial, and
* SNUM is not a superkey.

To deal with it, we can decompose S(SNUM, PNUM, SNAME, QUANTITY) into

(1) SUPPLIER(SNUM, SNAME) with

SNUM -> SNAME  
SNAME -> SNUM

with two candidate keys:

1. SNUM
2. SNAME

(2) SUPPLY(SNUM, PNUM, QUANTITY)  with

SNUM, PNUM -> QUANTITY.

Both are in BCNF.

***Example:***

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

A -> B,  B -> C, C -> A and C -> D.

There are three candidate keys:

1. A
2. B
3. C

Since every left hand side of any non-trivial functional dependency is a superkey,  R is in BCNF.

E.g.1 R(A,B,C,D) {A->B, C->D}: Highest NF: 1NF

CK: (1) AC; prime: AC, non-prime: BD  
A (a part of a CK) ->B (non-prime): violates 3NF

E.g.2 R(A,B,C,D) {A->B, AB->C, C->D, D->C}: Highest: 2NF

Canonical cover: {A->BC, C->D, D->C}:  
A+: ABCD  
L (left)/NR (not in RHS): A (must in every CK); R: B (not in any CK); M: CD (may be in CK)  
Consider: A, AC, AD, ACD  
CK: (1) A; prime: A; non-prime: BCD  
Checking FD:  
A (a CK, SK)->B (non-prime): ok to BCNF  
A (a CK, SK)->->C (non-prime): ok to BCNF  
C (not a part of a CK; not a proper subset of a CK; not a SK)->D (non-prime): ok with 2NF; not ok with 3NF  
D (not a part of a CK; not a SK)->C (non-prime): ok with 2NF; not ok with 3NF

|  |  |
| --- | --- |
| **Normal Form** | **Violating Non-trivial FD X-> Y** |
| 2NF | [1] X is a proper subset of a CK, and [2] Y is non-prime |
| 3NF | [1] X is not a SK, and [2] Y is non-prime |
| BCNF | X is a SK |

E.g.3

R(A,B,C) {A->C}: 1NF

L/NR: AB (Every CK must have both A and B)  
R: C (not in any CK)  
M: empty

CK: (1) AB.

A (a part of a CK: AB) 🡪 C (non prime): violates 2NF

E.g.

**Motivation of BCNF**

* The purpose of BCNF is to eliminate any unnecessary redundancy that functional dependencies can create in a relation.
  + In a BCNF relation, no value can be predicted from any other attributes besides the keys, using only functional dependencies.
  + This is because in a BCNF relation, using functional dependencies only,
    - any value can only be determined by a superkey,
    - but the superkey is unique.
  + However, there are other type of dependencies.
  + Therefore, there are higher normal forms.

***Example***

Consider the relation R(CITY, ZIP, STREET) again  
         
Using the code for the postal office, we have

CITY STREET -> ZIP, and ZIP -> CITY.

Hence, there are two candidate keys:

1. CITY STREET, and
2. ZIP STREET

Therefore, R is not in BCNF since in ZIP -> CITY, ZIP is not a superkey.

However, if we decompose R into two relations, each with two attributes, then the functional dependency

CITY STREET -> ZIP is lost (i.e. cannot be enforced within a single relation)

Therefore, we better leave the relation alone.

* Sometimes it is not desirable to achieve BCNF ==> need a less strict normal form (3NF).

**Normalization Theory Using Functional Dependencies**

* To use the theory on functional dependency:
  1. For a relation of a set of attributes, we analyze the assumptions of the applications.
  2. From the assumptions, we obtain the functional dependencies.
  3. Simplify the FD by finding a canonical cover.
  4. We determine the candidate keys and prime attributes.
  5. If the relation is not in BCNF, we perform decomposition.
  6. If BCNF cannot be satisfied, we aim for 3NF.

***Example:***

[1] Consider R(A, B, C, D, E) with

F = {BC->D, A->C, C->BD, AD->E} equivalent to {A->CE, C->BD}

(a) What are A+, B+, C+, D+ and E+?

(b) What are the candidate keys? Why?

(c) Show all prime attributes and non-prime attributes?

(d) Give a canonical cover of F?

(e) What is the highest normal form (up to BCNF) of R? Why?

(f) If R is not in BCNF, can you provide a lossless FD preserving decompositions of R into BCNF relations?

**Solution:**

(a) A+=ABCDE, B+=B, C+= BCD, D+=D, E+=E

(b) The candidate key is A

(c) Prime: A, non-prime: BCDE

(d) {A->CE, C->BD}

(e) 2NF since C->D violates 3NF: D is non-prime and C is not a Sk.

(f) Yes, the decomposition to R(A,C,E) {A->CE} and R2(B,C,D) {C->BD} satisfy the requirement.

[2] Consider R(A, B, C, D, E, F) with

F = {CD->E, A->BD, AC->EF. C->BD, F->E, EF->D}

(a) What are A+, B+, C+, D+, E+, F+?

(b) What are the candidate keys? Why?

(c) Show all prime attributes and non-prime attributes?

(d) Give a canonical cover of F?

(e) What is the highest normal form (up to BCNF) of R? Why?

(f) If R is not in BCNF, can you provide a lossless FD preserving decompositions of R into BCNF relations?

**Solution:**

(a) A+=ABD, B+=B, C+=BCDE, D+=D, E+=E, F+=DEF

(b) The candidate key is AC

(c) Prime: AC; non-prime: BDEF

(d) {A->BD, AC->F, C->BDE, F->DE}

(e) 1NF. A->B violates 2NF as A is a part of the a CK and B is non-prime.

(f) Yes, the following decomposition satisfies the requirement:

R1(B,C,D,E) {C->BDE}  
R2(D,E,F) {F->DE}  
R3(A,B,D) {A->BD}  
R4(A,C,F) {AC->F}

**3. Decomposition**

* Decomposition is a major tool for constructing relations satisfying high enough normal forms.
* Decomposition should be disciplined:
  + More relations may be less efficient in storage.
  + More relations may be less efficient in executing queries.
  + Some decompositions are harmful:
    1. Lossy decompositions.
    2. Decompositions that do not *preserve dependencies*.
* Hence, it is important to have lossless dependency-preserving decomposition.

**Lossy Decomposition**

***Example:***

Consider the relation EMP(EMP\_NO, DEPT\_NO, MANAGER\_NO) with

EMP\_NO ->  DEPT\_NO  
DEPT\_NO (not a CK) ->  MANAGER\_NO (non-prime): violates 3NF

Note that we do not have MANAGER\_NO -> DEPT\_NO in this example, since a manager can manage more than one departments under the assumptions made for this example.

|  |  |  |
| --- | --- | --- |
| **EMP\_NO** | **DEPT\_NO** | **MANAGER\_NO** |
| 12345 | ACCT | 90000 |
| 12399 | HR | 90000 |
| 30000 | ENG | 98000 |

The relation is not in BCNF because of the FD

DEPT\_NO -> MANAGER\_NO

Suppose we decompose the relation into

EMP1(EMP\_NO, MANAGER\_NO)  
DEPT(DEPT\_NO, MANAGER\_NO)  
  
The common attribute is MANAGER\_NO. They are obtained by projections from EMP:

EMP1:

|  |  |
| --- | --- |
| EMP\_NO | **MANAGER\_NO** |
| 12345 | 90000 |
| 12399 | 90000 |
| 30000 | 98000 |

DEPT:

|  |  |
| --- | --- |
| DEPT\_NO | **MANAGER\_NO** |
| ACCT | 90000 |
| HR | 90000 |
| ENG | 98000 |

If we do not lose any information by the decomposition, we should get the original relation from the natural join.

However,  EMP1 |x| DEPT <> EMP (lossy) is

|  |  |  |
| --- | --- | --- |
| **EMP\_NO** | **DEPT\_NO** | **MANAGER\_NO** |
| 12345 | ACCT | 90000 |
| *12345* | *HR* | *90000 extraenous and incorrect.* |
| *12399* | *ACCT* | *90000* |
| 12399 | HR | 90000 |
| 30000 | ENG | 98000 |

This is not the same as the original relation EMP. Spurious tuples were incorrectly created.

Hence, the decomposition of EMP(EMP\_NO, DEPT\_NO, MANAGER\_NO) into  
   
EMP1(EMP\_NO, MANAGER\_NO) and  
DEPT(DEPT\_NO, MANAGER\_NO)

is lossy.  It is not a good decomposition.

Example: A lossy decomposition using the supply database:

supply(snum, pnum, quantity) {snum, pnum} -> quantity

decomposed into:

s1(snum, quantity)  
s2(pnum, quantity)

select \*  
from (select snum, quantity from supply) as s1  
    natural join  
    (select snum, pnum from supply) as s2;

**Lossless Decomposition**

Example:

Consider now the following decomposition of EMP(EMP\_NO, DEPT\_NO, MANAGER\_NO):

EMP2(EMP\_NO, DEPT\_NO)  and  
EMP3(EMP\_NO, MANAGER\_NO)

The common attribute is EMP\_NO. We have EMP2 and EMP3:

EMP2:

|  |  |
| --- | --- |
| **EMP\_NO** | **DEPT\_NO** |
| 12345 | ACCT |
| 12399 | HR |
| 30000 | ENG |

EMP3:

|  |  |
| --- | --- |
| **EMP\_NO** | MANAGER\_NO |
| 12345 | 90000 |
| 12399 | 90000 |
| 30000 | 98000 |

Hence, EMP2 |x| EMP3:

|  |  |  |
| --- | --- | --- |
| **EMP\_NO** | **DEPT\_NO** | **MANAGER\_NO** |
| 12345 | ACCT | 90000 |
| 12399 | HR | 90000 |
| 30000 | ENG | 98000 |

This is exactly the same as the original relation EMP.  Therefore, the decomposition does not lose any information.  It is a lossless decomposition.

**Theory of Lossless Decomposition**

***Example:***

Why is the decomposition of EMP(EMP\_NO, DEPT\_NO, MANAGER\_NO) into

(1) EMP1(EMP\_NO, MANAGER\_NO) and DEPT(DEPT\_NO, MANAGER\_NO) lossy, and

(2) EMP2(EMP\_NO, DEPT\_NO) and EMP3(EMP\_NO, MANAGER\_NO) lossless?

**Theorem**: Suppose R(X, Y, Z) is decomposed into two relations R1(X, Y) and R2(X, Z).  X is the set of common attributes in R1 and R2.  The decomposition is lossless if and only if

(a) X -> Y (X is a CK in R1), or  
(b) X -> Z (X is a CK in R2).

***Example:***

In case (1), X is MANAGER\_NO, Y is EMP\_NO, Z is DEPT\_NO.

Neither condition (a) not (b) is satisfied.  Hence, (1) is lossy.

In case (2), X is EMP\_NO, Y is DEPT\_NO, Z is MANAGER\_NO.

Both conditions (a) and (b) are satisfied.  Hence, (2) is lossless.

* For decompositions into more than two relations, use the chase matrix algorithm (EN Algorithm 16.3).

***Example:***

Consider R(A,B,C,D,E) with {A->BC, CD -> E, BA -> C, D->B}.

It is decomposed into R1(A,B), R2(A,C), R3(C,D,E) and R4(B,E).

Chase algorithm: initial state -> change state. In an acceptable state: yes; otherwise: no if no more progress. (decidability problem).

Step 1. Create a table of 5 columns (number of columns) and 4 rows (number of component relations). Populate it with b(i,j).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Relation** | **A** | **B** | **C** | **D** | E |
| R1 | b(1,1) | b(1,2) | b(1,3) | b(1,4) | b(1,5) |
| R2 | b(2,1) | b(2,2) | b(2,3) | b(2,4) | b(2,5) |
| R3 | b(3,1) | b(3,2) | b(3,3) | b(3,4) | b(3,5) |
| R4 | b(4,1) | b(4,2) | b(4,3) | b(4,4) | b(4,5) |

Step 2. For each relation Ri, set all attribute Aj that appears in Ri from b(i,j) to a(j).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Relation** | **A** | **B** | **C** | **D** | E |
| R1 | a(1) | *a(2)* | b(1,3) | b(1,4) | b(1,5) |
| R2 | *a(1)* | b(2,2) | *a(3)* | b(2,4) | b(2,5) |
| R3 | b(3,1) | b(3,2) | *a(3)* | *a(4)* | *a(5)* |
| R4 | b(4,1) | *a(2)* | b(4,3) | b(4,4) | *a(5)* |

It is decomposed into R1(A,B), R2(A,C), R3(C,D,E) and R4(B,E).

FD: {A->BC, CD -> E, BA -> C, D->B}.

Step 3. While changes can be made with a FD X-> Y, with two rows in the table having the common X values in the following manner:

for every attribute W in Y:

* If one cell is an a and the other cell is an b, change the b to the a.
* If both cells are b's, change them to the same b.

Note that a specific FD can be applied more than once.

Applying A-> BC:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Relation** | **A** | **B** | **C** | **D** | **E** |
| R1 | a(1) | a(2) | *a(3)* | b(1,4) | b(1,5) |
| R2 | a(1) | *a(2)* | a(3) | b(2,4) | b(1,5) |
| R3 | b(3,1) | b(3,2) | a(3) | a(4) | a(5) |
| R4 | b(4,1) | a(2) | b(4,3) | b(4,4) | a(5) |

Applying CD -> E: no change since no two rows has the same values in CD.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Relation** | **A** | **B** | **C** | **D** | **E** |
| R1 | a(1) | a(2) | a(3) | b(1,4) | b(1,5) |
| R2 | a(1) | a(2) | a(3) | b(2,4) | b(2,5) |
| R3 | b(3,1) | b(3,2) | a(3) | a(4) | a(5) |
| R4 | b(4,1) | a(2) | b(4,3) | b(4,4) | a(5) |

Applying BA -> C: no change since R1 and R2 already have the same a's value: a(3).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Relation** | **A** | **B** | **C** | **D** | **E** |
| R1 | a(1) | a(2) | a(3) | b(1,4) | b(1,5) |
| R2 | a(1) | a(2) | a(3) | b(2,4) | b(2,5) |
| R3 | b(3,1) | b(3,2) | a(3) | a(4) | a(5) |
| R4 | b(4,1) | a(2) | b(4,3) | b(4,4) | a(5) |

Applying D->B: no change. No D's have the same value.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Relation** | **A** | **B** | **C** | **D** | **E** |
| R1 | a(1) | a(2) | a(3) | b(1,4) | b(1,5) |
| R2 | a(1) | a(2) | a(3) | b(2,4) | b(2,5) |
| R3 | b(3,1) | b(3,2) | a(3) | a(4) | a(5) |
| R4 | b(4,1) | a(2) | b(4,3) | b(4,4) | a(5) |

In fact, no FD can be applied again to change the matrix.

Step 4. If there is a row with only a's, the decomposition is lossless. Otherwise, there is no row with only a's and the decomposition is lossy.

Since there is no row with only a's, the decomposition is lossy.

***Example:***

Now suppose that C->DE is also in the FDs. That is, we have:

R(A,B,C,D,E) with {A->BC, CD -> E, BA -> C, D->B, *C->DE*}.

We will now have one more step.

Applying C->DE:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Relation** | **A** | **B** | **C** | **D** | **E** |
| R1 | a(1) | a(2) | a(3) | *a(4)* | *a(5)* |
| R2 | a(1) | a(2) | a(3) | *a(4)* | *a(5)* |
| R3 | b(3,1) | b(3,2) | a(3) | a(4) | a(5) |
| R4 | b(4,1) | a(2) | b(4,3) | b(1,4) | a(5) |

Now we have two rows with only a's and thus the decomposition is lossless.

**Dependency-Preserving Decomposition**

***Example:***

For the relation EMP(EMP\_NO,DEPT\_NO,MANAGER\_NO) with

EMP\_NO ->  DEPT\_NO  
DEPT\_NO ->  MANAGER\_NO,

The decomposition of EMP into

EMP2(EMP\_NO, DEPT\_NO)  and  
EMP3(EMP\_NO, MANAGER\_NO)

is lossless but does not preserve dependencies:

the FD  DEPT\_NO -> MANAGER\_NO

cannot be enforced by any relation after the decomposition. No relation contains both attributes.

For example, if we add the information EMP 23000 work in the ACCT department under manager 97000 and are not careful, we may have:

EMP2:

|  |  |
| --- | --- |
| **EMP\_NO** | **DEPT\_NO** |
| 12345 | ACCT |
| 12399 | HR |
| 30000 | ENG |
| ***23000*** | ***ACCT*** |

EMP3:

|  |  |
| --- | --- |
| **EMP\_NO** | **MANAGER\_NO** |
| 12345 | 90000 |
| 12399 | 90000 |
| 30000 | 98000 |
| *23000* | *97000* |

The FD  DEPT\_NO ->  MANAGER\_NO is violated.

Thus, for the relation EMP(EMP\_NO,DEPT\_NO,MANAGER\_NO) with

EMP\_NO ->  DEPT\_NO  
DEPT\_NO ->  MANAGER\_NO,

the best decomposition is into

EMP1(EMP\_NO, DEPT\_NO)  and  
DEPT(DEPT\_NO, MANAGER\_NO) { DEPT\_NO ->  MANAGER\_NO}

It is easy to show that, the decomposition is lossless, preserves dependencies, and that EMP1 and DEPT are both in BCNF.

* It is possible to decompose a relation such that
  1. all member relations are in 3NF,
  2. the decomposition is lossless, and
  3. all FDs are preserved.
* It is also possible to decompose a relation such that
  1. all member relations are in BCNF, and
  2. the decomposition is lossless, but
  3. not all FDs may be preserved.

**Algorithm for decomposition in 3NF relations**

* See Algorithm 16.6 of EN: lossless FD preserving decomposition into relations in 3NF.

***Example:***

Consider R(A,B,C,D,E) with F = {A->BC, CD -> E, ~~BA -> C,~~ D->B}.

Step 1. Find a canonical cover (as opposed to a minimal cover in EN) G for F.

The FD BA->C is redundant.

G = {A->BC, CD -> E, D->B} is a canonical cover.

L/NR: AD; R: BE; M: C  
AD+: ADBCE = R => CK: (1) AD; prime: AD; non-prime: BCE  
Analyze FD:

A (a part of a CK) ->B (non-prime): violates 2NF  
A->C  
CD ->E  
D->B

Highest NF: 1NF.

Step 2. For every FD X->Y in G, create a relation with the schema XY and add it to the result D. => preserve FD; also relations in 3NF or BCNF.

G = {A->BC, CD -> E, D->B} is a canonical cover.

Relations created:

R1(A,B,C) with A->BC  
R2(C,D,E) with CD->E  
R3(B,D) with D->B

This step ensures that all FDs are preserved.

Step 3. If no relation in D contains *a* candidate key of R, create a new relation with a candidate key of R being the schema and add it to the result D.

There is only one candidate key of R: AD. Since none of R1, R2 and R3 contains both A and D, create the relation

R4(A,D) with no FD: in BCNF => ensure lossless decomposition.

Step 4. Simplify D by removing relations that are redundant (i.e. that its schema is a subset of the schema of another relation).

No action as there is no redundant relation.

The result relations are all in BCNF.

***Example:***

Consider R(A,B,C,D,E) with {A->BCD, BC->D, D->C}: Highest NF: 1NF

Using the algorithm,

(1) Canonical cover: {A->BC, BC->D, D->C}  
  
L/NR: AE  
CK: (1) AE; prime: AE; non-prime BCD

D->C violates 3NF; ok with 2NF  
A (a proper subset of a CK)->B (non-prime) violates 2NF

(2) The following relations are created:

R1(A,B,C) with {A-> BC},  
R2(B,C,D) with {BC->D, D->C},  
R3(C,D) with {D->C}

(3) There is only one candidate key AE. Since it is not in any of R1, R2 or R3, R4 is created.

R4(A,E)

(4) R3(C,D) is removed as redundant since it is a subset of R2.

As the result, we have:

R1(A,B,C) with {A-> BC}, in BCNF  
R2(B,C,D) with {BC->D, D->C}, in 3NF but not in BCNF. CK: (1) BC, (2) BD. D->C violates BCNF.  
R4(A,E) with {}, in BCNF

* Algorithm 16.5 of EN is an algorithm for lossless decomposition into BCNF but FD may not be preserved.
* Sometimes, it is not possible to decompose a relation into two relations losslessly and preserve all FD, just to achieve BCNF.

***Example:***

Consider the relation R(A, B, C) with A -> B and C -> B.

R is not in 2NF.  It is not possible to decompose R into two relations losslessly while preserving all functional dependencies.

However, it is possible to decompose into three relations losslessly and with all functional dependencies preserved:

R1(A, B),  
R2(B, C) and  
R3(A, C).

**Introduction to XPath**

by K. Yue

**1. Introduction**

**Resources**:

* XPath 1.0: <http://www.w3.org/TR/xpath>
  + Still many examples in the Web.
  + Example: <http://www.whitebeam.org/library/guide/TechNotes/xpathtestbed.rhtm>
* XPath 2.0: <http://www.w3.org/TR/xpath20/>
  + Not quite the same as XPath 1.0.
  + Probably currently the most popular version.
* XPath 3.0: <http://www.w3.org/TR/xpath-30/>:
  + Generally a superset of XPath 2.0
* XQuery and XPath functions:
  + 1.0: <http://www.w3.org/TR/xquery-operators/>.
  + 3.0: <http://www.w3.org/TR/2014/REC-xpath-functions-30-20140408/>.
* EditiX (free edition): <http://free.editix.com/>
  + Free and good enough for our course.

**Basics:**

* XPath is used to *address* parts of an XML document.
* XPath is a W3C recommendation.
* The newest version is 3.0, which is largely backward compatible.
* XPath is used by XPointer, XSLT and XQuery.
* XPath is designed to access elements, but not creating new elements.
* Designed to be embedded in a host language, such as XSLT or XQuery.
* XQuery is a superset of XPath.
* The result is a *sequence*, which is an ordered list of items. (XPath 1.0: result: NodeSet.)
* An item can be a node or of atomic value.
* There are *7 node types*:
  1. Document: represent an entire XML document
  2. Element
  3. Attribute
  4. Comment
  5. Text
  6. Processing Instruction
  7. Namespace

**2. Path Expression**

* XPath uses *path expressions* to address parts of the documents, called *location path*.

C:\Bun\1\_F2021\_DBMS\website\demo\abc.html (absolute) in Windows folder.

.. parent directory

Abc.html (relative path, relative to current working directory.)

Unix: \ (root directory)

* A location path is composed of a sequence of location steps, separated by a '/': document root.
* A location path can be absolute or relative.
* An *absolute*location path starts with '/', the document root.
* A *relative* location path does not start with '/'. Its path is relative to a *context node*.
* This is similar to the Unix directory system.

***Example:***

Consider [film.xml](http://dcm.uhcl.edu/yue/courses/csci5333/current/notes/xml/film.xml) (with data extracted from Sakila).

//films/film

Film element that is a child of a films element that is a descendant of the document root.

* The XPath expression lists all film elements with a parent films elements in the XML document.
* It is an absolute location path.
* In Editix, use ">View > Windows > XPath View" to execute XPath expressions. You should select XPath 2.0 instead of XPath 1.0.

**Location path requirements:**

* A location step is composed of three parts:
  1. a *node axis* (required): to describe direction for navigation.
  2. a *node test* (required): to specify the node type, and
  3. a set of *node predicate* (optional): to specify additional inclusion test.

***Example:***

//films/child::film[actor/@id='162']

//films/child::film[child::actor/attribute::id='162']

//films/film[actor/@id='162']

Consider the location step:

child::film[actor/@id='162']

1. Node axis: child (default)
2. Node test: film
3. Node predicate: [actor/@id='162']

The XPath expression lists all <film> elements that:

1. are a child node of a <films> element in the document, and
2. have a child <actor> which has an attribute id with the value of '162'.

Note that actor/@id='162' is a *relative* path, relative to the *context* node, which is a film node.

**Node Axes:**

* An axis is the first part of the location step and is followed by :: before the node test and predicates.
* It indicates the direction to go for the next location step.
* There are 13 axes. Note the classification of axes into 'forward axis' and 'reverse axis'.
* In general, forward axes are preferable.

From XPath 3.0:

[40]      ForwardAxis      ::=      ("child" "::")  
| ("descendant" "::")  
| ("attribute" "::")  
| ("self" "::")  
| ("descendant-or-self" "::")  
| ("following-sibling" "::")  
| ("following" "::")  
| ("namespace" "::")  
[43]      ReverseAxis      ::=      ("parent" "::")  
| ("ancestor" "::")  
| ("preceding-sibling" "::")  
| ("preceding" "::")  
| ("ancestor-or-self" "::")

* The default axis is child.

**Node test:**

* Node test is the second part of a location step.
* It is required.
* There are three kind of node tests:
  1. NameTest: the name of an element or attribute node.
  2. NodeType test:
     + node(): all nodes, including comments and PI, excluding attributes and the document root.
     + text(): a text node
     + comment(): a comment node
  3. processing-instruction('pi-name')
* \* is a wildcard character matching any name. It is a name test.

**Node Predicate:**

* Predicate tests are the last part of a location step.
* They are enclosed by [] and are optional.
* There may be more than one predicate tests.
* XPath expressions and built-in functions can be used to construct predicate (boolean) expression as the added condition for inclusion.
* Boolean operators can be used: and, or.

**Shorthand:**

* . is the shorthand for self::node()
* .. is the shorthand for parent::node().
* // is the shorthand for /descendant-or-self::node()/
* @ is the shorthand for attribute::

***Example:***

//text()  
  
or

/descendant::\*/text()

list all text nodes.

Note the difference of

//actor/@id[.='20']  
//actor[@id='20']

* The first expression returns a *sequence* of id attributes.
* The second expression returns a sequence of actor elements.

//film/actor [position()=2]  
  
or  
  
//film/actor[2]

* The XPath function returns the second item in the *dynamic context of evaluation*.
* Thus, this returns the second child <actor> element of a <film> element.

**3. Sequence Expressions**

* In XPath 1.0, node set is the main data type of the returned result.
* In XPath 2.0, sequence is the main data type of the returned result.
* A *sequence* is an *ordered heterogeneous* collection of *items*.
* An item can be a node or an atomic value, but not a sequence.
* A sequence may contain *duplicate* atomic values or nodes.

***Example:***

(1, 5 to 8, "Bun Yue", 2.1)

(1+2, 5)

(1 to 50)[. mod 3 = 1]

//film/\* | //film

(1, 2, (3, (4, 5))) is (1,2,3,4,5)

* XPath 2.0 results are sequences. Atomic values are considered to be sequences with a single item.

**4. Other Expressions**

* XPath 2.0 supports many types of expressions not supported in XPath 1.0.

**Primary expressions**

* Includes literals (constants), variable references, function calls, use of parenthesis, and the context node (i.e.: .).
* There are many built-in functions.
* Functions may use the namespace fn.
* See <http://www.w3.org/TR/xpath-functions/>.

***Example:***

//film[count(actor)>=10]

or

//film[count(./actor)>=10]

returns all <film> nodes with 10 or more actors.

//film/actor [position()=2]  
//film/actor [fn:position()=2]  
  
or  
  
//film/actor[2]

* The XPath function returns the second item in the *dynamic context of evaluation*.
* Thus, this returns the second child <actor> element of a <film> element.

//film[starts-with(title/text(),'A')]

gives all <film> element with titles started with 'A'.

distinct-values(//film/actor[starts-with(text(),'A')])

gives a sequence of actor names starting with an 'A'.

**Arithmetic, Comparison and Logical Expressions**

* Similar to some of the operators found in other languages.
* However, XML Schema data types are quite different to most other languages. So be careful.

**For Expression and Variable Binding:**

* The Let expression allows the definitions of multiple *variables*and subsequently using them later in the expression returned.
* Format: Let $var1 := (expression-1), $var2 := (expression-2), ... return (expression)
* This allows the binding of expressions to variables for multiple uses.
* This is especially useful in XQuery.
* The For expression allows the definitions of multiple variables to *iterate* through sequence expressions then subsequently using them later in the expression returned.

Example:

for $film in (//film) return $film/actor

is the same as:

//film/actor

* There is actual no need to use the for expression in the example above.

List all names of all actors appeared in more than 35 films by using the for expression in XPath.

fn:distinct-values(//film/actor[for $a in . return count(//film/actor[@id = $a/@id]) > 35]/text())

This can be slow.

**Conditional Expressions:**

* If-then-else statement.

Example:

if (//film[title/text()='ADAPTATION HOLES']) then 'found Holes' else 'no Holes'

**Qualified Expressions**

* The existential and universal qualifiers.

Example:

All film elements with an actor of id of 4 or less

//film[some $a in actor satisfies $a/@id <= 4]

same as:

//film[actor/@id <= 4]

All film elements with *only* actors of id > 150.

//film[every $a in actor satisfies $a/@id > 150]

For [filmActor.xml](http://dcm.uhcl.edu/yue/courses/csci5333/current/notes/xml/filmActor.xml):

//film[every $a in //film[@id=937]/actorIds/actorId/@actorId satisfies ./actorIds/actorId/@actorId = $a]  
  
returns all film elements that have all actors who appearred in film with id 937.