**11/13/2019**

**Normalization**

by K. Yue

**1. Introduction**

* *Normal forms*: a set of rules to check for poor database design.
* Require the concepts of various kinds of *data dependency*: dependency or restrictions between *two sets of attributes*.
	1. functional dependency (FD, most important: up to BCNF): simple and practical
	2. multi-valued 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.

Venn’s Diagram



**General Overview**

* In general, the higher the normal forms a relation is in, the better the design of the relation in terms of avoiding unnecessary redundancy and inconsistency.
* However, it may be necessary to consider other issues, especially performances.
	1. Higher normal forms may be achieved by *decomposition*, resulting in more relations.
	2. More joins may then be needed to provide the data for a query, decreasing performance.
* 1NF is usually assumed. (1NF: No multi-valued column) However, there are relations not in 1NF in both theory and practice.
	1. For an 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 are 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 sets of attributes:
	+ Many to one (0..\* to 0..1): FD
	+ Many to many (0..\* to 0..\*)
* These relationships are the results of *assumptions* we made about the application requirements.

***Example***

Many to one relationships.

For *many* applications, the relationship between SSNum and NAME are many to one in a relation R(..,SSNum, Name, ...)

SSNum   (P)     ->     Name (Q)
(many)                  (one): every SSNUM is associated with only one Name.

Many: every name is associated with many SSNum.

P -> Q (given a P value, there is only one Q value.)

**Assumptions:**

1. A SSN uniquely identifies a person.
2. Given a SSNum, there can only be one Name associated with it (not allowing 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.

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.

***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.

FALL 2018 HW #7

[2] Consider the following table: Grade. Grade(StudentId, StudentFName, StudentLame, ClassId, InstructorId, Grade) The table stores the grade information of a student (identified by StudentId) taking a class (identified by ClassId). A class is always taught by a single instructor (identified by InstructorId).

(a) Identify the functional dependencies (FD) of the relation.

Attributes: StudentId, StudentFName, StudentLame, ClassId, InstructorId, Grade

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **StudentId** | **SFN** | **SLN** | **ClassId** | **InId** | **Grade** | **EnrollId** |
| S1 | Bun | Yue | T33351 | F123 | B | 1 (s) |
| S1 | Bun | Yue | T12892 | F123 | C+ | 2 |
| S1 | ~~Ben~~  | ~~Hur~~ |  |  |  |  |
| S2 | Ben | Hur | ~~T33351~~ | ~~F452~~ |  |  |
| S2 | Ben | Hur | T33351 | F123 | A- | 3 |
| ~~S1~~ | ~~Bun~~ | ~~Yue~~ | ~~T33351~~ | ~~F123~~ | ~~A~~ | Violates (t) {StudentId, ClassId} as a CK. |
|  |  |  |  |  |  |  |

Current instance does not violate Grade -> StudentId.
Grade –X-> StudentId based on assumption.

Row #3 no no, StudentId has unique name (assumption StudentId uniquely identify a student who has a unique name to be stored.)

Row #4 no no, same classId -> same InId (assumption: a class is taught by only one instructor)

Row #6 no no: a student taking the same class has only grade.

Row #3:
StudentId -> SFN, SLN

Row #4: a classId determines a class. A class has one and only one instructor (identified by InId)
ClassId -> InId

Row #6:
StudentId, ClassId -> Grade

 *functional dependency* X -> Y;

X: {StudentId, ClassId}

Y : {Grade}

(b) What are the candidate keys?

(K) StudentId, ClassId -> StudentId, ClassId, SFN, SLN, InId, Grade, EnrollId (R: relation schema)

(K) Enroll -> StudentId, ClassId, SFN, SLN, InId, Grade, EnrollId

Candidate Keys:

(1) {StudentId, ClassId} composite candidate key (CK)
(2) EnrollId: simple surrogate key serves as the primary key.

{StudentId, ClassId}: CK, SK

* is unique -> identify a unique row.
* Minimal ->
	+ {StudentId} (proper subset of {StudentId, ClassId}) is not unique; StudentId –X-> R (StudentID -> StudentID, SFN, SLN)
	+ {ClassId} (proper subset of {StudentId, ClassId}) is not unique; ClassId -> ClassId, InId.

For {StudentId, ClassId, SFN}

1. Unique
2. Not minimal: {StudentId, ClassId} (a proper subset of {StudentId, ClassId, SFN}) -> R

SK examples:

* {StudentId, ClassId}: SK, also a CK
* {StudentId, ClassId, SFN}: SK, not a CK
* ENROLLID: SK, also a CK
* {ENROLLID, CLASSID}: SK, not a CK

(c) What are the non-prime attributes?

Candidate Keys:

(1) {StudentId, ClassId} composite candidate key (CK)
(2) EnrollId: simple surrogate key serves as the primary key.

Attributes not in any CK.

Grade(StudentId, StudentFName, StudentLame, ClassId, InstructorId, Grade, EnrollId)

Non-prime attributes: SFN, SLN, INId, Grade

Prime (key) attributes: in some CKs: StudentId, CLassId, EnrollId

**Definition of FD:**

* A relation scheme R is said to *satisfy* the *functional dependency* X -> Y if for any relation r that uses R, if there are two tuples s and t in r such that s[X] = t[X], then s[Y] = t[Y].
* Same value of X implies same value of Y.

***Example***: This instance 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’ | 2 | 2000 violates FD 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***:

DEPT\_NO -> MANAGER\_NO:

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

CourseId, StudentId, Semester -> Grade

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

CourseId, StudentId, Year, Semester -> Grade

**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 p*hysical 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*.

**Some properties of Functional Dependency**

* Transitivity: X-> Y and Y->Z => X->Z (X: classId, Y: InId, Z: InstLName)
* Augmentation: X->Y => XA -> YA (X: classId, Y: InId, A studentId => classId, StudentId -> InId, StudentId)
* Reflexivity: if Y is a subset of X, then X-> Y ({classId, SFN} -> SFN
* Union: X->Y and X->Z => X->YZ X: StudentId, Y: SFN, Z-> SLN
* Decomposition: X->YZ => X->Y and X->Z

Example:

AB -> C => A-> B? {StuId, ClassId} -> Grade => StuId -> Grade? No

***Example***

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

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

By transitivity, EMP\_NO -> MANAGER\_NO
By union rule, EMP\_NO -> EMP\_NO, DEPT\_NO, MANAGER\_NO
By augmentation, EMP\_NO, MANAGER\_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.

X+ = every attribute determined by X.

***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.

* 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.

**Finding 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 is O(en).

To find all candidate keys for a set of FD, F:

1. Additional Material: Find the *canonical cover*, FC, first. This simplifies F. (This step is optional.)
2. Use heuristics to cut down the number of sets of attributes to check.
3. Use classification of attributes into three groups
	1. If X does not appear in the RHS of any f in FC, every candidate key must include X.
	2. 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. 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.

Q1. R(A,B,C,D): CK: (1) AB (unique and minimal)

How many superkeys? AB, ABC (unique and not minimal), ABD (unique and not minimal), ABCD (unique and not minimal): 4

M = # of attributes in R.

One CK: N attributes.

M =4, N = 2

# of superkey = 2 \*\* (M-N) = 4

Q2. R(A,B,C,D,E,F) CK: (1) AB

# of superkeys:

AB (+ or – C), (+ or – D) (+ or – E) (+ or – F)

E.g. ABD is SK

# of SK = s \*\* 4 = 16

Q3. R(A,B,C,D): CK: (1) A, (2) B

# of superkeys: 12

A, AB, AC, AD, ABC, ABD, ACD, ABCD (contains A)

B, BC, BD, BCD (Contains B but not A)

***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:

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}

**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.
* 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.

***Example***

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

|  |  |  |  |
| --- | --- | --- | --- |
| **DEPT\_NO** | **MANAGER\_NO** | **EMP\_NO** | **NAME** |
| D123 | 54321 | 10000, 12000, 13000Not atomic | Lady Gaga, Eminem, Lebron James ot atomic |
| D225 | 42315 | 21000, 22000 | Rajiv Gandhi, Bill Clinton |
| D337 | 33323 | 31000 | John Smithson |

The corresponding relation with 6 tuples is in 1 NF: atomicity related to requirements. i.e. Singled-valued.

|  |  |  |  |
| --- | --- | --- | --- |
| **DEPT\_NO** | **MANAGER\_NO** | **EMP\_NO** | **NAME** |
| D123 | 54321 | 10000 | Lady Gaga |
| D123 | 54321 | 12000 | Eminem |
| D123 | 54321 | 13000 | Lebron James (may be considered not atomic) |
| 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.
* It will be difficult to join using OSSkills.
* 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.
* A prime attribute appears in a 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.
* 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)

|  |  |  |  |
| --- | --- | --- | --- |
| **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 -> Credit: violate 2NF
2. Course, Student (Full CK) -> Grade (non-prime): not violating 2NF

Not appear in RHS: Course, Student (every CK should have Course, Student)
Only in RHS: Credit, Grade (Credit, Grade not in any CK)

CK: (1) Course, Student

Thus,

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

Course (part of a CK) -> Credit (non-prime)

To convert to 2NF, decompose Enroll into

1. Enroll(Course [FK], Student, Grade): CK: (1) Course, Student
2. Class(Course [PK], Credit): CK: Course

***Example from Hoffer (Partial):***

Invoice(OrderId, OrderDate, ProductId, ProductName, Quantity)

FD: An order maybe for a purchase of multiple products

1. OrderId (a part of a CK) -> OrderDate (non-prime): violate 2NF
2. ProductId (a part of a CK) -> ProductName (non-prime): violate 2NF
3. OrderId, ProductId (a full CK) -> Quantity (non-prime): does not violate 2NF

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OI** | **OD** | **PI** | **PN** | **Q** |
| O1 | 11/13/2019 | P1 | SO and So | 3 |
| O1 | 11/13/2019 | P2 | X and Y | 2 |
| O1 | 11/13/2019 | P3 | Ice cream | 5 |
| O2 | 11/11/2019 | P2 | X and Y | 1 |

CK: (1) OrderId, ProductId

Prime attributes: OrderId, ProductId

Non-prime attributes: OrderDate, ProductName, Quantity

FD 1 and 2 violate 2NF

To convert to 2NF, decomposition:

1. Order(OrderId, OrderDate) with {OrderId -> OrderDate}
2. Product(ProductId, ProductName) with {ProductId -> ProductName}
3. OrderLine(OrderId, ProductId, Quantity) {OrderId, ProductId -> Quantity}

Order(OrderId, OrderDate) with {OrderId -> OrderDate}

|  |  |
| --- | --- |
| **OI** | **OD** |
| O1 | 11/13/2019 |
| O2 | 11/11/2019 |

Product(ProductId, ProductName) with {ProductId -> ProductName}

|  |  |
| --- | --- |
| **PI** | **PN** |
| P1 | SO and So |
| P2 | X and Y |
| P3 | Ice cream |

OrderLine(OrderId, ProductId, Quantity) {OrderId, ProductId -> Quantity}

|  |  |  |
| --- | --- | --- |
| **OI** | **PI** | **Quantity** |
| O1 | P1 | 3 |
| O1 | P2 | 2 |
| O1 | P3 | 5 |
| O2 | P2 | 1 |

CK: {OrderId, ProductId}

E.g. R(A,B,C) {A->B, BC->A}

C in every CK (because C not in any RHS)
A and B in both LHS and RHS (A, B may be in a CK.)

Check:

C: C+ = C
CA; CA+ = CAB (CA is a CK)
CB; CB+ = CBA (CB is a CK)
CAB: not minimal.

CK: (1) CA, (2) CB

Prime attributes: C, A, B

Non-prime attributes: no

2NF only check for non-prime attribute in RHS.

A (a part of a CK) -> B (prime): does not violate 2NF.

BC(full CK) ->A (prime): does not violate 2NF.

In 2NF

**Third Normal Form**

* (New definition) A relation R is said to be in the third normal form if for every *non-trivial* functional dependency X -> A,
	1. X is a superkey, *or*
	2. A is a *prime* (key) attribute.
* (Old definition) A relation R is in 3NF if
	1. R is in 2NF, and
	2. There is no *transitive* dependency of *nonkey* attributes on the candidate keys.
* 3NF cannot eliminate all redundancy due to functional dependencies.

X->A violates 3NF if (1) X is not a superkey, and (2) A is non-prime.

E.g. R(A,B,C) {A->B, BC->A}

In 3NF because all attributes are prime.

***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 -> NAME, DEPT\_NO
	2. DEPT\_ NO (not a SK) -> MANAGER\_NO (non-prime attribute): violates 3NF

DEPT\_NO+ = DEPT\_NO, MANAGER\_NO

* then
	1. There is only one candidate key: EMP\_NO (only simple CK -> in 2NF)
	2. Prime attributes: EMP\_NO
	3. Non-prime attributes: NAME, DEPT\_NO, MANAGER\_NO.
	4. The relation is in 2NF.
* The relation is not in 3NF because of
	1. (old definition): the FD EMP\_NO -> MANAGER\_NO can be deduced from transitivity via the non-prime attribute DEPT\_NO.
	2. (new definition):
		+ EMP\_NO is the only candidate key.
		+ EMP\_NO is prime
		+ DEPT\_NO and MANAGER\_NO are non-prime.
		+ DEPT\_NO -> MANAGER\_NO violates 3NF.

***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. Given a supplier and a part, the quantity is unique.
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 -> 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.

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 |

***Example***

Consider the relation R(CITY, STREET, ZIP) with the FDs:

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

There are two candidate keys:

1. CITY STREET, and
2. ZIP STREET

Hence, all attributes are prime attributes and the relation is in both 2NF and 3NF.

* 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*.

***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.

**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, 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)

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 possible for a relation to be in BCNF ==> need a less strict normal form (3NF).

**Normalization Theory Using Functional Dependencies**

* To use the theory on functional dependency:
	+ For a relation of a set of attributes, we analyze the assumptions of the applications.
	+ From the assumptions, we obtain the functional dependencies.
	+ We determine the candidate keys and prime attributes.
	+ If the relation is not in BCNF, we perform decomposition.
	+ If BCNF cannot be satisfied, we aim for 3NF.

***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:

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

CK:  {SupplierId, ProductId, ArrivalTime}

Non-prime attributes: SupplierName, ProductDesc, Quantity

Highest Normal Form: 1NF

SupplierId -> SupplierName violates 2NF since SupplierId is a part of a candidate key and Quantity is non-prime.

**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:
		- *Lossy* decompositions.
		- 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, MGR\_NO) with

EMP\_NO ->  DEPT\_NO
DEPT\_NO ->  MGR\_NO

Note that we do not have MGR\_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** | **MGR\_NO** |
| 12345 | ACCT | 90000 |
| 12399 | HR | 90000 |
| 30000 | ENG | 98000 |

The relation is not in BCNF because of the FD

DEPT\_NO -> MGR\_NO

Suppose we decompose the relation into

EMP1(EMP\_NO, MGR\_NO)
DEPT(DEPT\_NO, MGR\_NO)

The *common attribute* is MGR\_NO. They are obtained by projections from EMP:

EMP1:

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

DEPT:

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

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

However,  EMP1 |x| DEPT is

|  |  |  |
| --- | --- | --- |
| **EMP\_NO** | **DEPT\_NO** | **MGR\_NO** |
| 12345 | ACCT | 90000 |
| *12345* | *HR* | *90000* |
| *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, MGR\_NO) into

EMP1(EMP\_NO, MGR\_NO) and
DEPT(DEPT\_NO, MGR\_NO)

is *lossy*.  It is not a good decomposition.

**Lossless Decomposition**

Example:

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

EMP2(EMP\_NO, DEPT\_NO)  and
EMP3(EMP\_NO, MGR\_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** | **MGR\_NO** |
| 12345 | 90000 |
| 12399 | 90000 |
| 30000 | 98000 |

Hence, EMP2 |x| EMP3:

|  |  |  |
| --- | --- | --- |
| **EMP\_NO** | **DEPT\_NO** | **MGR\_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 loss any information.  It is a *lossless*decomposition.

**Theory of Lossless Decomposition**

***Example:***

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

(1) EMP1(EMP\_NO, MGR\_NO) and DEPT(DEPT\_NO, MGR\_NO) *lossy*, and

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

**Theorem**: Suppose R(X, Y, Z) is decomposed into 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, *or*
(b) X -> Z.

***Example:***

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

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

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

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

* For decompositions into more than two relations, use the chase matrix algorithm, which is not covered in this course.

**Dependency-Preserving Decomposition**

***Example:***

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

EMP\_NO ->  DEPT\_NO
DEPT\_NO ->  MGR\_NO,

The decomposition of EMP into

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

is lossless but does not *preserve dependencies*:

the FD  DEPT\_NO -> MGR\_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** |
| 12345 | ACCT |
| 12399 | HR |
| 30000 | ENG |
| ***23000*** | ***ACCT*** |

EMP3:

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

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

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

EMP\_NO ->  DEPT\_NO
DEPT\_NO ->  MGR\_NO,

the best decomposition is into

EMP1(EMP\_NO, DEPT\_NO)  and
DEPT(DEPT\_NO, MGR\_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
	+ all member relations are in 3NF,
	+ the decomposition is lossless, and
	+ all FDs are preserved.
* It is also possible to decompose a relation such that
	+ all member relations are in BCNF, and
	+ the decomposition is lossless, but
	+ not all FDs may be preserved.

**Algorithm for decomposition in 3NF relations (not covered)**

* There are many algorithms for decomposition.
* In particular, the following example shows the step of an lossless, FD preserving algorithm that guarantees 3NF.
* Since we do not study canonical cover in this course, step 1 may be hard.

***Example:***

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

Step 1. Find a *canonical cover* G for F. (Loosely speaking, an equivalence of F with the least number of FD and attributes)

The FD BA->C is redundant.

G = {A->BC, CD -> E, D->B}.

Step 2. For every FD X->Y in G, create a relation with the schema XY and add it to the result D.

Relations created:

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

It can be seen very easily that R1, R2 and R3 are all in 3NF. Furthermore, 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 A, create the relation

R4(A,D) with no FD

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}

Using the algorithm,

(1) Canonical cover: {A->BC, BC->D, D->C}

(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.

As in 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
R4(A,E) with {}, in BCNF

* There are other decomposition algorithms.
* 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).