**CSCI 4333.2 Fall 2024**

**11/18/2024**

**[6] (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 candidate key:**

* **AB, ABC, ABD, ABE, ABCD, ABCE, ABDE and ABCDE are superkeys: 8**

|  |
| --- |
| **R(A,B,C,D)**    **A is CK => superkey: A, AB, AC, AD, ABC, ABD, ACD, ABCD**  **ABC is a CK => superkey: ABC, ABCD** |

2nd CK: C

C: C, CA, CB, CD, CE, CAB, CAD, CAE, CBD, CBE, CDE, CABD, CABE, CADE, CBDE, CABDE (16)

Maximum: 8 + 16 – 4 = 20

2nd CK: ACDE

[4] (20%) Consider R(P, Q, R, S, T, U) with F = {RS->PQ, QR->S, S->P, U->ST, STU->R, Q->U} (a) What are P+, Q+, R+, S+, T+, U+? (b) What are the candidate keys? (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?

[4] For F = {RS->PQ, QR->S, S->P, U->ST, STU->R, Q->U}

(a)

P+=P

Q+= PQRSTU

R+ = R

S+ = SP

T+ = T

U+ = PQRSTU

(b) CK: [1] Q, [2] RS and [3] U.

(c) Prime: Q,R,S,U; non-prime: P,T

(d) Canonical Cover: there are many, for examples, {RS-> Q, Q->RSTU, S-> P, U->Q}

(e) Highest NF: 1NF. S->P violates the 2NF.

(f) The decomposition into

R1(Q,R,S,T,U) { Q->RSTU}

~~RA(R,S,Q) {RS -> Q}~~

R2(S,P) {S->P}

~~RB(U, Q) {U->Q}~~

R1(Q,R,S,T,U) {RS-> Q, Q->RSTU, S-> P, U->Q} and

R2(P,S) {S-> P} are lossless

and FD preserving. R1 and R2 are in BCNF. Further decomposition of R1(Q,R,S,T,U) is also acceptable but

not desirable.

**Normal Forms and Theory of Normalization**

by K. Yue

**1. Normal Forms Using Functional Dependencies**

**1.1 First Normal Form**

* A relation is in 1NF if all attribute values are *atomic*: no repeating group, no composite attributes, no internal structures.
* Semantically, an attribute is atomic if it cannot be broken down to smaller pieces with individual meanings.
* Theoretically, a relation may only have atomic attributes.  Thus, all pure 'relations' satisfy 1NF.
* In practice, DBMS may allow data types with composite values and internal structures, e.g. set, JSON, spatial, XML, etc.

***Example***

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

* The value "10000, 12000, 13000" of the field EmpIds can be broken into three components, "10000", "12000", and "13000" with individual meanings.
* The same is true for the field Names.
* Note the plural forms of EmpIds and Names.

|  |  |  |  |
| --- | --- | --- | --- |
| **DeptId** | **ManagerId** | **EmpIds** | **Names** |
| D123 | 110 | 10000, 12000, 13000 | Lady Gaga, Eminem, Lebron James |
| D225 | 440 | 21000, 22000 | Rajiv Gandhi, Bill Clinton |
| D337 | 300 | 31000 | John Smithson |

An alternate design of the relation in 1NF is shown below. The following instance has six rows.

|  |  |  |  |
| --- | --- | --- | --- |
| **DeptId** | **ManagerId** | **EmpId** | **Name** |
| D123 | 110 | 10000 | Lady Gaga |
| D123 | 110 | 12000 | Eminem |
| D123 | 110 | 13000 | Lebron James |
| D225 | 440 | 21000 | Rajiv Gandhi |
| D225 | 440 | 22000 | Bill Clinton |
| D337 | 300 | 31000 | John Smithson |

* Why atomic?
  + Relational theory and operations treat attributes as atomic.
  + No need to worry about the correctness and consistency of internal structures.
* Relations not satisfying 1NF have undesirable 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 other tables using OSSkills.
* Data entry problems and issues, e.g. Linux linux, linx, etc., may further degrade data quality and introduce inconsistency.
* On the other hand, relations may be in Non-First Normal Form (NFNF of NF2), mainly for *performance* considerations. E.g., NoSQL DB.

**1.2 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.
* Review: A prime attribute (also called key attribute) appears in one or more candidate keys. Otherwise, it is a non-prime (non-key) attribute. Note that a relation may have many candidate keys.
* There is *no partial dependency* in 2NF.

***Example***

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

Enroll(CourseId, Credit, StudentId, Grade): R(A,B,C,D) {AC->D, A->B): not in 2NF; A (a part of a CK, CourseId)-> B (credit, non-prime)

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

We assume the following FD.

1. CourseId -> Credit: violates 2NF
2. CourseId, StudentId (full CK) -> Grade (non-prime) (ok with 2NF)

Thus,

1. {CourseId, StudentId} is the only candidate key.
2. Prime attributes: CourseId, StudentId
3. Non-prime attribute: Credit, Grade.
4. FD (1), CourseId (a part of/a propose subset of a CK)-> Credit (non-prime), is a partial dependency that violates 2NF.
   * CourseId (a proper subset of a CK) -> Credit (a non-prime attribute)

To convert to relations in 2NF, decompose Enroll into

1. Enroll(CourseId, StudentId, Grade) {CourseId, StudentId -> Grade}; R1(A,C,D) {AC -> D}
2. Class(CourseId, Credit) {CourseId -> Credit}; R2(A,B) {A->B}: CK: A

Both tables are in 2NF.

1. {CourseId -> Credit} violates 2NF in Enroll(CourseId, Credit, StudentId, Grade)
   * CourseId is a proper subset of a CK (CourseId, StudentId) in Enroll
2. {CourseId -> Credit} does not violate 2NF in Class(CourseId, Credit)
   * CourseId is a CK in Class. It is not a proper subset of a CK.

***Example from Ricardo:***

NewClass(courseNo, stuId, stuLastName, facId, schedule, room, grade). We have:

courseNo, stuId -> grade  
stuId -> stuLastName  
courseNo -> facId, schedule, room

StuId -> stuLastName and courseNo -> facId, schedule, room violate 2NF

To convert to 2NF, decomposition:

1. Course(courseNo, facId, schedule, room) { courseNo -> facId, schedule, room } The FD is no longer violating 2NF in the new table Course since courseNo is a CK in Course.
2. Student(stuId, stuLastName) { StuId -> stuLastName } The FD is no longer violating 2NF in the new Student table since StuId is a CK in Student
3. Enroll(courseNo, stuId, grade) { courseNo, stuId -> grade }

**1.3 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: included for historical reason. Do not use it for normalization analysis.) A relation R is in 3NF if
  1. R is in 2NF, and
  2. There is no *transitive* dependency of *non-prime* attributes on the candidate keys.
* Review: A superkey S of a relation R satisfies S -> R (uniqueness). It does not need to satisfy the minimal property.
* 3NF can identify unnecessary redundancy that 2NF cannot identify.
* 3NF still cannot eliminate all redundancies due to functional dependencies.

***Example***

* The following relationis in 2NF, but is not in 3NF.

|  |  |  |  |
| --- | --- | --- | --- |
| **DeptId** | **ManagerId** | **EmpId** | **Name** |
| *D123* | *110* | 10000 | Lady Gaga |
| *D123* | *110* | 12000 | Eminem |
| *D123* | *110* | 13000 | Lebron James |
| D225 | 205 | 21000 | Rajiv Gandhi |
| D225 | 205 | 22000 | Bill Clinton |
| D337 | 333 | 31000 | John Smithson |

* If we assume the following canonical set of FDs:
  1. EmpId -> Name, DeptId: A-> BC: ok with 2NF and 3NF
  2. DeptId -> ManagerId; C (not a SK)->D (non-prime): violates 3NF; ok with 2NF
* then
  1. There is only one candidate key: EmpId: A (If all CK are simple => in 2NF)
  2. Prime attribute: EmpId: A
  3. Non-prime attributes: Name, DeptId, ManagerId. B,C,D
  4. The relation is in 2NF.
* The relation is not in 3NF because:
  1. EmpId is the only candidate key.
  2. EmpId is prime.
  3. DeptId and ManagerId are non-prime.
  4. DeptId -> ManagerId violates 3NF because:
     1. DeptId is not a SK.
     2. ManagerId is non-prime.
* To resolve, decompose the relation into:
  1. Department(DeptId, MangaerId) { DeptId -> ManagerId }, R1(C,D) {C -> D}: ok; CK in R1: C; in 3NF
  2. Employee(EmpId, Name, DeptId) { EmpId -> Name, DeptId }, R2 (A,B,C) {A -> BC} in 3NF

***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.

Note that a relation such as Employee(EmePId, Name, Street, City, Zip, State) is not in 3NF.

This is a classical example you can find in many database textbooks. Note that the two FDs may not actually be correct in the United States.

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

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

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

***Example***

Consider the relation

S(SId, PId, SName, Quantity) with the following assumptions:

1. SId 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 FD:

1. SId -> SName
2. SName -> SId
3. SId PId -> Quantity
4. SName PId -> Quantity

Note that SId and SName are *equivalent*.

The candidate keys are:

1. SId PId
2. SName PId

Prime attributes: SId, PId, SName

Non-prime attribute: Quantity.

The candidate keys are:

1. SId PId
2. SName PId

Prime attributes: SId, PId, SName

Non-prime attribute: Quantity.

The relation is in 3NF. Note:

1. SId (not a SK) -> SName does not violate 3NF as SName is prime. Violates BCNF
2. SName -> SId does not violate 3NF as SId is prime. Violates BCNF
3. SId PId -> Quantity does not violate 3NF as {SId, PId} is a CK and also a SK.
4. SName PId -> Quantity does not violate 3NF as {SName, PId} is a CK and also a SK.
5. In 3NF

However, there are unnecessary redundancy.

|  |  |  |  |
| --- | --- | --- | --- |
| ***SId*** | ***SName*** | **PId** | **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 |

Thus, 3NF does not detect all design problems using FD.

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

SId -> SName is

1. non-trivial, and
2. SId is not a superkey.

To deal with it, we can decompose S(SId, PId, SName, Quantity) into

(1) Supplier(SId, SName) with

SId -> SName  
SName -> SId

with two candidate keys:

1. SId
2. SName

(2) Supply(SId, PId, Quantity)  with

SId, PId -> 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.

**1.5 Checking Highest Normal Form by Violations**

To find the highest normal form for a relation R, check every non-trivial FD X->Y of R for violation.

* Note that in the table below, A is a single attribute. Use the decomposition rule if necessary. For example, for AB->CD, check AB->C and AB->D.

|  |  |
| --- | --- |
| **Normal Form's*Violation*** | **Non-trivial FD X -> A** |
| 2NF | (1) X is a proper subset of a candidate key of R, and (2) A is a non-prime attribute. |
| 3NF | (1) X is not a superkey of R, and (2) A is a non-prime attribute. |
| BCNF | X is not a superkey. |

* If there is no violation of a normal form, then R is in that normal form.
* If there is one violation of a normal form, then R is not in that normal form.

E.g. [d] R(A,B,C,D) {AB->C, AD->C}

L/NR: A B D  
M:  
R: C  
CK: (1) ABD  
prime attributes: A, B,D

Non-prime: C

AB (a proper subset of a CK) -> C (non-prime) violates 2NF

AD (a proper subset of a CK) -> C (non-prime) violates 2NF

Highest NF: 1NF

R1(A,B,C) {AB -> C) BCNF  
R2(A,C,D) {AD -> C) BCNF  
R3(A,B,D) {} BCNF

***Example:***

Consider R(A,B,C,D) {A->B, B->AC, C->D}

Using decomposition rule, we have {A->B, B->A, B->C, C->D}

We find two CK: [1] A, [2] B  
Prime attributes: A, B  
Non-prime attributes: C, D

Checking for violation:

|  |  |  |  |
| --- | --- | --- | --- |
| **FD** | **Ok with 2NF** | **Ok with 3NF** | **OK with BCNF** |
| A->B | Yes | Yes | Yes |
| B->A | Yes | Yes | Yes |
| B->C | Yes | Yes | Yes |
| C->D | Yes | No | No |

Thus, the highest NF is 2NF

**1.6 Motivation of BCNF**

* The purpose of BCNF is to eliminate any unnecessary redundancy that FD can create in a relation.
  + In a BCNF relation, no value can be predicted from any sets of non-unique attributes, using *only* FD.
  + This is because in a BCNF relation, using FD only,
    - any attribute 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 FD

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

Therefore, we better leave the relation alone.

* Sometimes it is not possible for a relation to be in BCNF => need to settle in a less strict normal form (3NF).

**1.7 Normalization Theory Using Functional Dependencies**

* To use the theory on FD:
  1. For a relation of a set of attributes, we analyze the assumptions of the applications.
  2. From the assumptions, we obtain a set of FDs.
  3. Find a canonical cover of the set of FDs.
  4. Find all candidate keys, prime and non-prime attributes.
  5. If the relation is not in BCNF, we perform *decomposition*.
  6. 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

1. SupplierId is a part (proper subset) of a candidate key, i.e., {SupplierId} ⊂ {SupplierId, ProductId, ArrivalTime}, and
2. Quantity is non-prime.

Not that A ⊂ B means that A is a proper subset of B.

**2. Decomposition**

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

**2.1 Lossy Decomposition**

***Example:***

Consider the relation Emp(EmpId, DeptId, ManagerId) R(A,B,C) with (CK: A)

EmpId ->  DeptId; A->B  
DeptId ->  ManagerId; B (not a SK) ->C (non-prime) (violates 3NF)

Note that we do not have ManagerId -> DeptId in this example, since this organization allows a manager to manage more than one Departments. Note that ManagerId 90000 manages two Departments.

|  |  |  |
| --- | --- | --- |
| **EmpId** | **DeptId** | **ManagerId** |
| E1 | ACCT | M3 |
| E2 | HR | M3 |
| E3 | *ENG* | *M6* |
| E4 | *ENG* | *M6* |

The relation is not in 3NF because of the FD

DeptId (not a SK) -> ManagerId (non-prime)

Suppose we decompose the relation into

Emp1(EmpId, *ManagerId*)  
Dept(DeptId, *ManagerId*)  
  
The *common attribute* for the component relations is ManagerId. The relations are obtained by projections from Emp:

Emp1:

|  |  |
| --- | --- |
| **EmpId** | **ManagerId** |
| E1 | M3 |
| E2 | M3 |
| E3 | M6 |
| E4 | M6 |

Dept:

|  |  |
| --- | --- |
| **DeptId** | **ManagerId** |
| ACCT | M3 |
| HR | M3 |
| ENG | M6 |

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

However,  Emp1 |x| Dept <> Original table is

|  |  |  |
| --- | --- | --- |
| **EmpId** | **DeptId** | **ManagerId** |
| E1 | ACCT | M3 |
| *E1* | *HR* | *M3* |
| *E2* | *ACCT* | *M3* |
| E2 | HR | M3 |
| E3 | ENG | M6 |
| E4 | ENG | M6 |

This is not the same as the original relation Emp. Spurious rows are incorrectly included in the result.

Hence, the decomposition of Emp(EmpId, DeptId, ManagerId) into  
   
Emp1(EmpId, ManagerId) and  
Dept(DeptId, ManagerId)

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

**2.2 Lossless Decomposition**

Example:

Consider now the following decomposition of Emp(EmpId, DeptId, ManagerId):

Emp2(*EmpId*, DeptId)  and  
Emp3(*EmpId*, ManagerId)

The common attribute is EmpId. We have Emp2 and Emp3:

Emp2:

|  |  |
| --- | --- |
| **EmpId** | **DeptId** |
| E1 | ACCT |
| E2 | HR |
| E3 | ENG |
| E4 | ENG |

Emp3:

|  |  |
| --- | --- |
| **EmpId** | **ManagerId** |
| E1 | M3 |
| E2 | M3 |
| E3 | M6 |
| E4 | M6 |

Hence, Emp2 |x| Emp3:

|  |  |  |
| --- | --- | --- |
| **EmpId** | **DeptId** | **ManagerId** |
| E1 | ACCT | M3 |
| E2 | HR | M3 |
| E3 | ENG | M6 |
| E4 | ENG | M6 |

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

**Definition.** A decomposition is lossless if the natural joins of the component relations result in the original relation. Otherwise, it is lossy.

**2.3 Theory of Lossless Decomposition**

***Example:***

Why is the decomposition of Emp(EmpId, Dept, ManagerId) into

(1) Emp1(EmpId, ManagerId) and Dept(DeptId, ManagerId) *lossy*, and

(2) Emp2(EmpId, DeptId) and Emp3(EmpId, ManagerId) *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 ManagerId, Y is EmpId, Z is Dept.

Neither condition (a) nor (b) is satisfied.  Hence, (1) is lossy.

In case (2), X is EmpId, Y is DeptId, Z is ManagerId.

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.

**2.4 Dependency-Preserving Decomposition**

***Example:***

For the relation Emp(EmpId, DeptId, ManagerId) with

EmpId ->  DeptId  
DeptId ->  ManagerId,

The decomposition of Emp into

Emp2(EmpId, DeptId)  and  
Emp3(EmpId, ManagerId)

is lossless but it does not *preserve dependencies*:

the FD  DeptId -> ManagerId

cannot be assured within a single relation after the decomposition. No relation contains both attributes.

For example, if we add the information Emp E6 work in the ACCT Department under manager M9 carelessly, we may have the following table.

* Since ACCT has M3 as manager already, it cannot also have M9 as its manager.

Emp2:

|  |  |
| --- | --- |
| **EmpId** | **DeptId** |
| E1 | *ACCT* |
| E2 | HR |
| E3 | ENG |
| E4 | ENG |
| *E6* | *ACCT* |

Emp3:

|  |  |
| --- | --- |
| **EmpId** | **ManagerId** |
| E1 | *M3* |
| E2 | M3 |
| E3 | M6 |
| E4 | M6 |
| *E6* | *M9: should be M3* |

As a result, the FD  DeptId ->  ManagerId is violated.

* Department with DeptId ACCT has two ManagerId
  1. M3 (via EmpId E1)
  2. M9 (via EmpId E6)

Thus, for the relation Emp(EmpId, DeptId, ManagerId) with

EmpId ->  DeptId  
DeptId ->  ManagerId,

the best decomposition is

Emp1(EmpId, *DeptId*)  and  
Dept(*DeptId*, ManagerId)

It is easy to show that, the decomposition is lossless, preserves dependencies, and that Emp1 and Dept are both in BCNF.

***2.4.2 Decomposition Algorithms***

1. 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.
2. 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.

**2.5 Algorithm for decomposition into 3NF relations**

* There are many algorithms for decomposition.
* We will not cover the details of the algorithms, but they are illustrated by the example below.
* In particular, the following examples show the steps of an lossless, FD preserving algorithm that guarantees 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* G for F.

The FD BA->C is redundant.

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

We may perform normalization analysis to see whether decomposition is necessary.

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

We have: AD+ = AD BC E  
  
Thus, CK: [1] AD  
prime: A, D  
non-prime: B, C, E

Normalization analysis:

|  |  |  |  |
| --- | --- | --- | --- |
| **Non-trivial FD** | **2NF** | **3NF** | **BCNF** |
| A -> B: [1] A ⊂ AD, [2] A is not a SK, [3] B is non-prime | violate | violate | violate |
| A -> C: [1] A ⊂ AD, [2] A is not a SK, [3] C is non-prime | violate | violate | violate |
| CD -> E: [1] CD ⊄ AD, [2] C is not a SK, [3] E is non-prime | ok | violate | violate |
| D -> B: [1] D ⊂ AD, [2] D is not a SK, [3] B is non-prime | violate | violate | violate |

Thus, the highest normal form of R is 1NF. Decomposition is necessary.

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

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

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 and BCNF. 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. This step assures losslessness.

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

R4(A,D) with no FD: {}

Step 4. Simplify the decomposition 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}

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

CK: AE

A->BCD violates 2NF; A is a part of a CK; BCD are non-prime.

Using the algorithm,

(1) Canonical cover: {A->BC, BC->D, D->C}; A->D is removed since it is a redundant FD.

(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* BCNF relations losslessly and with all functional dependencies preserved:

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

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

R is not in BCNF.  It is not possible to decompose R into BCNF relations losslessly while preserving all FD.

**Introduction to MongoDB**

by K. Yue

**1. Introduction**

* NoSQL document model distributed database owned by MongoDB (NASDAQ: MDB).
* Documents are stored in JSON format.
* Three versions:
  + Community server: open source and version
  + Enterprise server: commercial version
  + Atlas: cloud version

A screenshot of a computer

Description automatically generated

**1.1 Installation**

For this class, install the followings.

1. MongoDB community server: ensure that it includes Mongo Compass, a MongoDB client, <https://www.mongodb.com/try/download/community>
2. Mongo Shell:
   1. mongosh.exe: a Javascript shell for interacting with MongoDB, <https://www.mongodb.com/try/download/shell>.
   2. Do *not* use mongo.exe, the deprecated former shell.
3. Mongo Compass includes a Mongosh.
4. MongoDB tools: command line utilities including import and export, <https://www.mongodb.com/try/download/database-tools>.
   1. After unzipping, you may put mongosh and these utilities in the same location of the other mongoDB programs, e.g., C:\Program Files\MongoDB\Server\5.0\bin.
   2. You may add the directory “C:\Program Files\MongoDB\Server\5.0\bin”, or similar, in the system PATH variable so these tools can be used anywhere.
5. To be able to use MongoDB through Python, you will to install a driver: "pip install pymongo" in cmd.

**1.2 Server-Client DBMS architecture**

* Like many DBMS, MongoDB uses a client server model.
* Server:
  + In case the MongoDB server has not been started, run "mongod" in a command terminal.
  + To check whether mongod is running, execute 'tasklist /FI "IMAGENAME eq mongod.exe"' in Command CLI.
  + It listens to a port to accept and interpret commands and return results.
  + mongod's default port: 27017.
* Clients: send MongoDB commands and accept results. Clients used in this course:
  + Mongo Compass

A screenshot of a computer

Description automatically generated

* + mongosh
  + Python through pymongo (if Python is used.)

**1.3 Resources**

* MongoDB manual: <https://docs.mongodb.com/manual/>

**2. MongoDB Structures**

* MongoDB is structured as db -> collection -> document (objects have nested structured) in a way similar to db -> table -> row in relational DB.
* Thus, documents are inserted into a collection of a db.
* db and collection do not need to exist before referencing them.
* In MongoDB's db, within mongosh:
  + 'use tinker' set the default db to tinker.
  + The keyword db refers to the default db.
  + If 'tinker' does not exist, it will be created.

**2.1 Using mongo command CLI through mongosh**

* Run 'mongosh' in command CLI in your working directory.
* Mongosh accept JavaScript commands in a mongo shell setting.
* For inserting documents, it supports two methods, insertOne and insertMany.
* See mongosh CRUD:<https://docs.mongodb.com/mongodb-shell/crud/insert/>.

**3. Writing to Mongo**

1. See CRUD operation in Mongo Guide to begin with: <https://docs.mongodb.com/guides/>.
   1. However, the guide uses the deprecated shell "mongo" instead of "mongosh".
   2. Since mongosh should be used, be mindful of discrepancies.

***Example:***

In mongosh, execute the code:

use tinker  
db.test1.insertOne(  
   {  
      "StudentId" :1,  
      "StudentName" : "Joseph Connor"  
   }  
)  
  
gives the following result:

test> use tinker  
switched to db tinker  
tinker> db.test1.insertOne (  
...     {  
.....           "StudentId" :1,  
.....           "StudentName" : "Joseph Connor"  
.....   }  
... )  
{  
  acknowledged: true,  
  insertedId: ObjectId("61e0d5f36753d9628bb4bfa1")  
}  
tinker> db.test1  
tinker.test1

Note:

1. In "db.test1.insertOne (", the '(' must not be put into the next line.
2. If not, mongosh thinks that the current JavaScript statement has ended and you may get:

tinker> db.test1.insertOne  
[Function: insertOne] AsyncFunction {  
  apiVersions: [ 1, Infinity ],  
  serverVersions: [ '3.2.0', '999.999.999' ],  
  returnsPromise: true,  
  topologies: [ 'ReplSet', 'Sharded', 'LoadBalanced', 'Standalone' ],  
  returnType: { type: 'unknown', attributes: {} },  
  deprecated: false,  
  platforms: [ 0, 1, 2 ],  
  isDirectShellCommand: false,  
  acceptsRawInput: false,  
  shellCommandCompleter: undefined,  
  help: [Function (anonymous)] Help  
}  
tinker> (  
...     {  
.....           "StudentId" :1,  
.....           "StudentName" : "Joseph Connor"  
.....   }  
... )  
{ StudentId: 1, StudentName: 'Joseph Connor' }  
  
In Windows, you may start Compass through the startup manual:

A screenshot of a computer

Description automatically generated

In Mongo Compass (you may enter nothing in the 'Paste your connection string' connect box):

A screenshot of a computer

Description automatically generated

* Note that a field \_id with a system generated object id is created. It is unique and can be served as an id.

If the code is executed one more time, Mongo Compass has:

A screenshot of a computer

Description automatically generated

Note:

1. There are now two Joseph Connor.
2. StuId is not a 'primary key'.
3. Document model is not set-theoretic. Relation model is set-theoretic.

To insert a document 'doc' only when it does not already exist, use something like:

if (db.test1.find(doc).count() == 0) { db.test1.insertOne(doc) }

Note:

1. 'db.test1.find(doc)' finds the documents doc (one document in the example below). It returns a cursor, which is an iterator of the query result.
2. cursor has a method count() to count the result.

The following session illustrates this concept.

Code:

show dbs  
db.dropDatabase()  
show dbs  
  
// remove tinker  
use tinker  
db.test1.find()  
doc = {  
      "StudentId" :1,  
      "StudentName" : "Joseph Connor"  
}  
doc  
if (db.test1.find(doc).count() == 0) { db.test1.insertOne(doc) }  
db.test1.find()  
if (db.test1.find(doc).count() == 0) { db.test1.insertOne(doc) }  
db.test1.find()

Session:

tinker> db.test1.find()  
  
tinker> doc = {  
...             "StudentId" :1,  
...             "StudentName" : "Joseph Connor"  
... }  
{ StudentId: 1, StudentName: 'Joseph Connor' }  
tinker> doc  
{ StudentId: 1, StudentName: 'Joseph Connor' }  
tinker> if (db.test1.find(doc).count() == 0) { db.test1.insertOne(doc) }  
{  
  acknowledged: true,  
  insertedId: ObjectId("61e0e49e6753d9628bb4bfa5")  
}  
tinker> db.test1.find()  
[  
  {  
    \_id: ObjectId("61e0e49e6753d9628bb4bfa5"),  
    StudentId: 1,  
    StudentName: 'Joseph Connor'  
  }  
]  
tinker> if (db.test1.find(doc).count() == 0) { db.test1.insertOne(doc) }  
  
tinker> db.test1.find()  
[  
  {  
    \_id: ObjectId("61e0e49e6753d9628bb4bfa5"),  
    StudentId: 1,  
    StudentName: 'Joseph Connor'  
  }  
]

**3.2 Unique Index**

* A unique index can be used to ensure that all documents within the collection must have unique values on the fields.
* This can be used for use cases of inserting the document only if the unique index has an unique value.
* Thus, a unique index can serve as a candidate key (if *it is not missing*) for identifying document in the collection.

***Example:***

Code:

// remove tinker  
show dbs  
db.dropDatabase()  
show dbs  
// create index  
db.test1.createIndex( { "StudentId": 1 }, { unique: true } )  
doc = {  
      "StudentId" :1,  
      "StudentName" : "Joseph Connor"  
}  
doc  
db.test1.insertOne(doc)  
db.test1.insertOne(doc)  
  
  
Session:

tinker> // create index  
  
tinker> db.test1.createIndex( { "StudentId": 1 }, { unique: true } )  
StudentId\_1  
tinker> doc = {  
...       "StudentId" :1,  
...       "StudentName" : "Joseph Connor"  
... }  
{ StudentId: 1, StudentName: 'Joseph Connor' }  
tinker> doc  
{ StudentId: 1, StudentName: 'Joseph Connor' }  
tinker> db.test1.insertOne(doc)  
{  
  acknowledged: true,  
  insertedId: ObjectId("6570fb99629ad72db73f7bcf")  
}  
tinker> db.test1.insertOne(doc)  
MongoServerError: E11000 duplicate key error collection: tinker.test1 index: StudentId\_1 dup key: { StudentId: 1 }

Note:

* In 'db.test1.createIndex( { "StudentId": 1 }, { unique: true } )', '"StudentId": 1' means the attribute is a part of the index. It does not mean the value of "StudentId" should be one. 1 stands for true here.
* In { unique: true }, the index is set to have the uniqueness property.

***Example:***

db.test1.insertMany([  
   {   "StudentId" :2,  
      "GPA": 3.72  
   },  
   {   "StudentId" :3,  
      "GPA": 1.69  
   },  
   {  
      "BCAssetId": "78c22fc6-5dec-11ec-bf63-0242ac130002",  
      "BCAssetType": "BCAssetTypeMetadata",  
      "BCAssetName": "BCAssetTypeMetadata: MBSEModel",  
      "ForBCAssetType": "MBSEModel",  
      "Version": {  
         "Version": "1.0",  
         "Subversion": null,  
         "StartTime": "2019-01-13T07:23:13+06:00"  
      }  
   }  
])  
db.test1.find()

Note:

1. The method insertMany() inserts many documents.
2. Documents may have *no*schema.
3. Within a collection, there can be many kinds of documents.
4. StudentId is a unique index, but it may not exist.
5. Thus, a Mongo's unique index is not exactly the same as a candidate key (which cannot be null) of a table in the relational model.

**4. Querying**

* Basically use the find method.
* find as supported in Mongosh: <https://docs.mongodb.com/manual/reference/method/db.collection.find/>.
* Format: db.collection.find(query, projection).

**4.1 Toyu**

Create the ‘toyu’ database in MongoDB.

1. Download the file: [toyu-db.gz](https://dcm.uhcl.edu/yue/courses/joinDB/Fall2024/notes/nosql/toyu-db.gz).
2. Ensure that you have download MongoDB tools: command line utilities including import and export, <https://www.mongodb.com/try/download/database-tools>.
3. Run the command:

mongorestore --archive="toyu-db.gz" --gzip --nsFrom='toyu.\*' --nsTo='toyu.\*'

Note that the design of toyu is not the typical way one would design a MongoDB. Instead, it is intended to look like the toyu MySQL database for ease of comparison.

***Example:***

[1] Show all students.

use toyu  
db.student.find()

Getting rid of \_id:

db.student.find({},  
   { "\_id": 0 }    
)

[2] // Show all information of students majoring in 'CINF'.  
  
db.student.find({"major": "CINF"},  
    { "\_id": 0 }  
)  
  
[3] Show all student names. Return an array of student objects.

db.student.find({},  
   { "fname": 1, "lname":1, "\_id": 0 }    
)

[4] Show all student names in this format:

student #0: Tony Hawk  
student #1: Mary Hawk  
student #2: David Hawk  
student #3: Catherine Lim  
student #4: Larry Johnson  
student #5: Linda Johnson  
student #6: Lillian Johnson  
student #7: Ben Zico  
student #8: Bill Ching  
student #9: Linda King

Solution:

result = db.student.find({},  
   { "fname": 1, "lname":1, "\_id": 0 }    
).toArray()  
  
// May not always work as toArray() returns a promise,  
// which may not be ready for use.  
result.forEach((x,i) => console.log('student #' + String(i) + ': ' + x["fname"] + ' ' + x["lname"]))

[5] Show the names and credits (ach) of students majoring in 'CSCI' and having 40 or more credits.

db.student.find(  
   { "major": "CSCI", "ach" : {$gte: 40} },  
   { "fname": 1, "lname":1, "ach":1, "\_id": 0 }    
)

Notes:

1. MongoDb's query and projection operators: <https://docs.mongodb.com/manual/reference/operator/query/>

[6] Show the first name and last name of students with a first name starting with a L or B, case insensitive.

db.student.find(  
   { "fname": { $regex: /^[lb]/, $options: "i" } },  
   { "fname": 1, "lname":1, "\_id": 0 }    
)

Notes:

1. A regular expression is used: <https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex>.
2. For regular expressions in general, see: <https://en.wikipedia.org/wiki/Regular_expression>
3. Explanations:
   1. ^: match the beginning of a string.
   2. [lb]: a character class that matches 'l', 'b' (and also 'L' and 'B' since case insensitive matching is used.)
   3. option a: case insensitive matching.

[7] Show the names and credits (ach) of students majoring in 'CSCI' and having 40 or more credits.

db.student.find(  
   { "$and": [ { "major": "CSCI"}, { "ach": {"$gte": 40}} ] },  
   { "fname": 1, "lname":1, "ach":1, "\_id": 0 }    
)

**4.2 Aggregation**

1. "Aggregation operations process multiple documents and return computed results."
2. See: <https://docs.mongodb.com/manual/aggregation/>.
3. It can be used to replace map-reduce functionality. See: <https://docs.mongodb.com/manual/reference/map-reduce-to-aggregation-pipeline/>.
4. There will not be programming questions on aggregation in the final examination.

[8] Show the number of faculty in each department.

In SQL:  
  
SELECT DISTINCT deptCode, Count(facId)  
FROM faculty  
GROUP BY deptCode;

In MongoDB:

db.faculty.aggregate([  
    {"$group" : {\_id:"$deptCode", "count":{$sum:1}}}  
])  
  
db.faculty.aggregate(  
   [    
      { $group: { "\_id": "$deptCode", "count": {$sum:1}} },  
      { $project: { "deptCode": "$\_id" , "num\_faculty": "$count",  "\_id": 0}}  
   ]  
)

Notes:

1. $group: form group.
2. $sum: aggregate function.

[9] Show the names of students who have enrolled in 10000: joining two document.

This should have the similar effect of the SQL statement:

SELECT DISTINCT s.fname, s.lname  
FROM student AS s, enroll AS e  
WHERE s.stuId = e.stuId AND e.classId = 10000;

In MongoDB:

db.student.aggregate([  
{$lookup:  
    {  
      from: "enroll",  
      let: {joinValue: '$stuId'},  
      pipeline: [  
           { $match:  
                 { $expr:  
                    { $and:  
                       [  
                         { $eq: [ "$stuId",  "$$joinValue" ] },  
                         { $eq: [ "$classId", 10000 ] }  
                       ]  
                    }  
                 }  
            }      
        ],  
        as: "enrollment"     }},  
  { $match: {"enrollment":  { $ne: [] }}},    
  { $project: { "fname": 1, "lname": 1, "\_id": 0}}   
])

Notes:

1. An 'join' example.
2. Joining is difficult in MongoDB than SQL as document database should not be designed like a relational database.
3. In particular:
   1. The relational model uses a flat structure with no embedment.
   2. The document model uses a hierarchical structure encouraging embedment.

**4.3 Running Javascript program not using mongosh**

Try run [tinker.js.txt](https://dcm.uhcl.edu/yue/courses/joinDB/Fall2024/notes/nosql/tinker.js.txt) (remove .txt when saving)

// run "npm i mongodb" in the working directory.  
  
// To run this program: node tinker1.js  
const mongo = require('mongodb');  
  
var MongoClient = mongo.MongoClient;  
var url = 'mongodb://localhost:27017';  
  
MongoClient.connect(url, function(err, client) {  
   db = client.db("toyu");  
   console.log("hello");  
   var result = db.collection("faculty").find(  
      { "rank": "Assistant Professor" },  
      { "fname": 1, "lname": 1, "deptCode": 1, "\_id": 0,  }    
   ).toArray()  
   result.then((docs) => {  
        console.log(docs);  
    }).catch((err) => {  
        console.log(err);  
    }).finally(() => {  
        client.close();  
    });  
});