**CSCI 4333 Section 1  
Annotation**

**Theory of Functional Dependency**

by K. Yue

**1. Armstrong's axioms**

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

***Example***

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

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

***Example:***

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

AB -> AB, AB->A, AB->B, AB-> Φ  
A -> A, A-> Φ  
B -> B, B-> Φ

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

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

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

Interpretation: If

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

then every Employee has only one manager.

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

***Example***

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

Proof:

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

**2. Keys and Superkeys Revisited**

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

***Example***

In Employee(EmpId, DeptId, ManagerId) with

EmpId -> DeptId and  
DeptId -> ManagerId.

By the transitivity axiom, EmpId -> ManagerId  
By the union rule, EmpId -> EmpId, DeptId, ManagerId  
By the augmentation axiom, EmpId, ManagerId -> DeptId, ManagerId  
                         
Hence, EmpId is a CK of Employee(EmpId, DeptId, ManagerId).

On the other hand,

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

Furthermore, there are four superkeys:

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

**3. Finding Candidate Keys**

**3.1 Closure of Attributes**

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

***Example***

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

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

A+ = AC  
B+ = ABCD = R  
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 subsets of ABCD to check for candidate keys:

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

Thus, B is the only CK.

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

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

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

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

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

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

***Example:***

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

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

We have:

L/NR: B (in every CK)  
M: A, D (may be in some CK)  
R: C (not in any CK)

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

B+: BACD

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

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

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

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

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

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

Note that

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

**4.2 Equivalence and cover**

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

**4.3 Canonical and Minimal Covers**

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

***Example***

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

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

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

***Example***

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

AB->C is redundant since for

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

Alternatively, we may state that

G |- AB-> C.

***Example***

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

Using decomposition rule,

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

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

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

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

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

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

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

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

***Example:***

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

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

No, Since in F, BD+ = BD

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

[2] F |- AE -> B ?

No, since AE+ = AE C

[3] Give a canonical cover for F.

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

[4] Show all candidate keys.

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

CK: [1] ABD, [2] CBD

***Example (Tedious):***

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

***Solution:***

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

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

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

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

B+ = B  
C+ = C

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

On the other hand, in F:

A+ = ABCDEF

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

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

Removing identical FD, we have F:

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

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

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

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

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

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

In F': we have

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

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

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

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

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

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

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

{BC -> AF, A-> BCDE}

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

{BC -> A, A-> BCDEF}

***Exercise:***

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

Find:

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

***Exercise:***

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

**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.
* Semantically, an attribute is atomic if it cannot be broken down to small 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, e.g. set, JSON, spatial, XML, etc.

***Example***

Consider the following table with 3 records. It is not in 1 NF. (nfnf: NON-FIRST NORMAL form)

* 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.
* 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 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.

**1.2 Second Normal Form**

* A relation R is in 2NF if
  1. R is in 1NF, and
  2. all *non-prime* (non-key) attributes are *fully* dependent on the *candidate* keys.
* A prime attribute (also called key attribute) appears in one or more candidate keys. Otherwise, it is a non-prime attribute. Note that a relation may have many candidate keys.
* A non-prime attribute (also called non-key attribute) does not appear in any candidate key.
* 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(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 |

We assume the following FD.

1. Course (a proper subset of a CK) -> Credit (non-prime): violate 2NF
2. Course, Student (a CK) -> Grade (non-prime): dose not violate 2NF.

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.
5. Highest NF: 1NF.

To convert to relations in 2NF, decompose Enroll into

1. Enroll(Course, Student, Grade) {Course, Student (full CK) -> Grade}
2. Class(Course, Credit) {Course (full CK) -> Credit}: CK: Course.

***Example from Ricardo:***

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

courseNo, stuId -> grade  
stuId (a proper subset of a CK) -> stuLastNam (non-prime)  
courseNo (a proper subset of a CK) -> facId, schedule, room (non-prime)

L/NR: coursed, stuId  
M:  
R: grade, stuLastName, facid, schedule, room

CK: [1] coursed, stuId  
prime: coursed, stuId  
Non-prime: grade, stuLastName, 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.) 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.
* 3NF can identify unnecessary redundancy that 2NF cannot identify.
* 3NF still cannot eliminate all redundancy due to functional dependencies.

***Example***

* The following relation *is* 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 (full CK, SK) -> Name, DeptId (non-prime): ok with 3NF
  2. DeptId (not a part/proper subset; no SK) of CK-> ManagerId (non-prime): violates 3NF.
* then
  1. There is only one candidate key: EmpId
  2. Prime attribute: EmpId
  3. Non-prime attributes: Name, DeptId, ManagerId.
  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 (CK, SK) -> ManagerId }: ok with 3NF
  2. Employee(EmpId, Name, DeptId) { EmpId (CK, SK) -> Name, DeptId }: ok with 3NF

***Example***

Consider the relation

S(SId, PId, SName, Quantity) with the following assumptions: in 3NF

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 (not a SK) -> SName (prime): ok with 3NF
2. SName (not a SK, a proper subset of a CK) -> Sid (prime): ok with 3NF
3. SId PId (full CK) -> Quantity (non-prime): ok with 3NF
4. SName PId -> Quantity

Note that SId and SName are *equivalent*.

* (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.

The candidate keys are:

1. SId PId
2. SName PId

Prime attributes: SId, PId, SName

Non-prime attribute: Quantity.

The relation is 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.

***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 redundancy 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***

Employee(EmpId, Name, DeptId, ManagerId) with

1. EmpId -> Name
2. EmpId -> DeptId
3. DeptId -> ManagerId

is not in BCNF.

The functional dependency  DeptId -> ManagerId

1. It is non-trivial, and
2. DeptId 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(EmpId, Name, DeptId, ManagerId) into

1. Emp(EmpId, Name, DeptId) with

EmpId -> Name, DeptId

and

2. Department(DeptId, ManagerId) with

DeptId -> ManagerId

Both relations are in BCNF since

* EmpId is a superkey of the relation Emp.
* DeptId 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(SId, PId, SName, Quantity) with the following non-trivial functional dependencies:

1. SId (not a SK) -> SName (violate BCNF)
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.

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 belowm, A is a single attribute. Use 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.

***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}

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

Canonical Cover: {A->B, B->AC, 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 (CK, SK)->B | Yes | Yes | Yes |
| B (CK, SK) ->A | Yes | Yes | Yes |
| B (CK, SK) ->C | Yes | Yes | Yes |
| C (not a proper subset of a CK; not a SK) ->D (non-prime) | Yes | No | No |

Thus, the highest NF is 2NF

|  |  |
| --- | --- |
| **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. |

F23 Q4:

[4] (20%) Consider the following relation R(A,B,C,D,E) {A->B, AB->D, AD->E, C->D}

Canonical Cover:

Extraneous attributes:

A+: A B D E   
B+: B  
D+: D   
  
{A -> B, A -> D, A-> E, C->D}

Redundant FD:

Check A->D redundant: {A -> B, ~~A -> D~~, A-> E, C->D} |- A->D? No

{A -> B, ~~A -> D~~, A-> E, C->D}: A+: A B E

{A -> B, A -> D, A-> E, C->D}: no redundant FD

A -> D is not redundant.

Canonical cover: {A->BDE, C->D}  
minimal cover: {A -> B, A -> D, A-> E, C->D}:

(a) Show all candidate keys.

Canonical cover: {A->BDE, C->D}  
  
L/NR: A, C  
M:  
R: B, D, E

CK: [1] AC

(b) What is the highest normal form (up to BCNF)? Why? 1NF  
prime: A, C  
non-prime: B, D, E  
Normalization analysis: {A -> B, A -> D, A-> E, C->D}:

|  |  |  |  |
| --- | --- | --- | --- |
| **FD** | **Ok with 2NF** | **Ok with 3NF** | **OK with BCNF** |
| A (a proper subset of a CK) -> B (non-prime) | NO | NO | NO |
| A (a proper subset of a CK) -> D (non-prime) | No | No | No |
| A (a proper subset of a CK) -> E (non-prime) | No | No | No |
| C (a proper subset of a CK) -> D (non-prime) | No | No | No |

|  |  |
| --- | --- |
| **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. |

(c) If it is not in BCNF, can you losslessly decompose R into component relations in BCNF while preserving functional dependencies?

Canonical cover: {A->BDE, C->D}

[1] One relation for each FD in CC. (remove violation; preserve FD)

R1(A,B,D,E) {A (CK, SK) -> BDE}: BCNF  
R2(C,D) {C (CK, SK) -> D}: BCNF

[2] Add a table with a CK of R if it is not in a component relation. CK: AC (lossless decomposition)  
  
R3(A,C) {}: BCNF

[3] Simplification: no action

Q: {~~A->B,~~ AB->D, AD->E, C->D}: { AB->D, AD->E, C->D}:

* A+: A  
  B+: B  
  C+: CD
* L/NR: A, B, C  
  M: D  
  R: E
* CK: ABC+: ABC D E  
  prime: A, B,C  
  non-prime: D, E
* AB (a proper subset of a CK -> D (non-prime): violates 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 other 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 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***

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.

**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) with

EmpId ->  DeptId  
DeptId (not a proper subset of a CK) ->  ManagerId (non-prime) violates #NF

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** |
| 12345 | ACCT | 90000 |
| 12399 | HR | 90000 |
| 30000 | ENG | 98000 |

The relation is not in BCNF because of the FD

DeptId -> ManagerId

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** |
| 12345 | *90000* |
| 12399 | *90000* |
| 30000 | 98000 |

Dept:

|  |  |
| --- | --- |
| **DeptId** | **ManagerId** |
| ACCT | *90000* |
| HR | *90000* |
| ENG | 98000 |

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

However,  Emp1 |x| Dept is

|  |  |  |
| --- | --- | --- |
| **EmpId** | **DeptId** | **ManagerId** |
| 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 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** |
| 12345 | ACCT |
| 12399 | HR |
| 30000 | ENG |

Emp3:

|  |  |
| --- | --- |
| **EmpId** | **ManagerId** |
| 12345 | 90000 |
| 12399 | 90000 |
| 30000 | 98000 |

Hence, Emp2 |x| Emp3:

|  |  |  |
| --- | --- | --- |
| **EmpId** | **DeptId** | **ManagerId** |
| 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.

**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, X is a CK of R1, *or*  
(b) X -> Z, X is a CK of R2.

***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 nManagerId) 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 23000 work in the ACCT Department under manager 97000 (incorrect) and are not careful, we may have:

Emp2:

|  |  |
| --- | --- |
| **EmpId** | **DeptId** |
| 12345 | ACCT |
| 12399 | HR |
| 30000 | ENG |
| ***23000*** | ***ACCT*** |

Emp3:

|  |  |
| --- | --- |
| **EmpId** | **ManagerId** |
| 12345 | 90000 |
| 12399 | 90000 |
| 30000 | 98000 |
| ***23000*** | ***97000*** |

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

* Department with DeptId ACCT has two ManagerId
  1. 90000 (via EmpId 12345)
  2. 97000 (via EmpId 23000)

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.

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 example shows the step 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** | **BCNR** |
| 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. (1) Improve NF. (2) Preserve FD

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

Relations created:

R1(A,B,C) with A (full CK in R1; a proper subset in R)->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.

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}

Using the algorithm,

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

L/NR: A, E  
M: B, C, D  
R:  
CK: [1] AE

(2) The following relations are created:

{A->BC, BC->D, D->C};

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 a 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: CK: [1] BC, [2] BD  
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).

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.

[5] (5%) It is known that for R(A,B,C,D,E): 1. R has exactly two candidate keys 2. A is a candidate key. 3. D and E are non-prime attributes. How many superkeys can R have?

[5] It is known that for R(A,B,C,D,E):

1. R has exactly two candidate keys

2. A is a candidate key.

3. D and E are non-prime attributes.

Possible CK: B, C, BC

There are three scenarios for the second candidate keys:

1. B => 24 SK {A, AB, AC, AD, AE, ABC, ABD, ABE, ACD, ACE, ADE, ABCD, ABCE, ABDE, ACDE, ABCDE, B, BC, BD, BE, BCD, BCE, BDE, BCDE}

2. C => 24 SK {You figure out}

3. BC => 20 SK {A, AB, AC, AD, AE, ABC, ABD, ABE, ACD, ACE, ADE, ABCD, ABCE, ABDE, ACDE, ABCDE, BC, BCD, BCE, BCDE}

Thus, the number SK: {20,24}

**Introduction to MongoDB**

by K. Yue

**1. Introduction**

* MongoDB is a NoSQL document model distributed database owned by MongoDB (NASDAQ: MDB).
* Documents are stored in JSON format (BSON to be exact).
* Three versions:
  1. Community server: open source version
  2. Enterprise server: commercial version
  3. Atlas: cloud version

**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, to 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
  + 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 in a way similar to db -> table -> row in MySQL.
* Thus, documents are inserted into a collection of 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 may use 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 value 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 some sort of 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. Elementary 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/Spring2024/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/>
2. $gte: the greater than or equal operator.

[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.
3. $project: pass along the requested fields to the next phase in a pipeline.

[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. $$joinValue: the value of the variable $joinValue, which is '$stuId of enroll.
4. 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 programs directly without using mongosh**

Try run [tinker1.js.txt](https://dcm.uhcl.edu/yue/courses/joinDB/Spring2024/notes/nosql/tinker1.js.txt) (remove .txt when saving)

// run "npm i mongodb" in the working directory.  
  
// To run this program: node tinker.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();  
    });  
});