Integrity Constraints and Functional Dependencies

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- Examples:
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

Integrity Constraints on a Single Relation

- **Domain constraints**
  - Specify that within each tuple, the value of each attribute must be an atomic value from the domain.
- **not null**
  - For example, if every STUDENT tuple must have a valid, non- NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL
- **primary key**
  - No primary key value can be NULL
  - Two distinct tuples in any state of relation cannot have identical values for (all) attributes in key
- **unique**
  - To specify candidate keys
- **check** (P), where P is a predicate
  - To ensure that a condition is satisfied

Domain constraints

- The domain salaryPerHour includes a constraint to ensure that the value of the salary per hour must be greater than 4.00€

  create domain salaryPerHour **numeric** (5,2)
  constraint checkSalary
  check (**value** > 4.00)

- We can also limit the values of the domain:

  create domain accountType **char** (10)
  constraint checkAccountType
  check (**value in** (‘Saving’, ‘Checking’**)**)

- And we can use queries in the check clause:

  check (branchName **in**
  (**select** branchName **from** branch))

Domain constraints

- Domains typically include:
  - Numeric data types for integers and real numbers
  - Characters
  - Booleans
  - Fixed-length strings
  - Variable-length strings
  - Money
  - Date, time, timestamp

Examples:

- **date**: Dates, containing a (4 digit) year, month and day
  - Example: **date** ‘2005-7-27’
- **time**: Time of day, in hours, minutes and seconds.
  - Example: **time** ‘09:00:30’ **time** ‘09:00:30.75’
- **timestamp**: date plus time of day
  - Example: **timestamp** ‘2005-7-27 09:00:30.75’

Not Null and Unique Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**
- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes *A*1, *A*2, … *A*m form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

The check clause

- **check** (P)

  where P is a predicate

Example: ensure that semester is one of **fall, winter, spring or summer**:

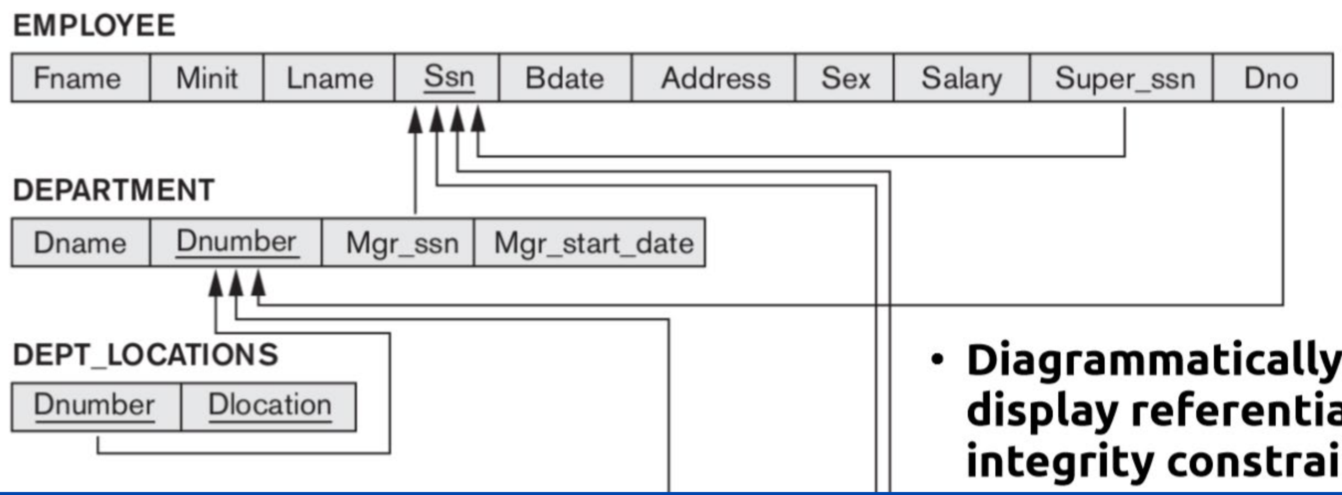
```
create table section (
    course_id varchar (8),
    sec_id varchar (8),
    semester varchar (6),
    year numeric (4,0),
    building varchar (15),
    room_number varchar (7),
    time slot id varchar (4),
    primary key (course_id, sec_id, semester, year),
    check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

Referential Integrity constraint

- Maintains consistency among tuples in **two relations**
- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology".
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S. They can also be NULL.
  - The attributes in FK must have the same domain(s) as the primary key attributes PK

Referential Integrity constraint

- Example: In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT relation.
- This means that a value of Dno in any tuple of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT (the Dnumber attribute) in some tuple of the DEPARTMENT relation, or the value of Dno can be NULL if the employee does not belong to a department or will be assigned to a department later.



- **Diagrammatically display referential integrity constraints**

Cascading Actions in Referential Integrity

- **create table** *course (*

  course_id **char**(5) **primary key**,
  title **varchar**(20),
  dept_name **varchar**(20) **references** *department*
)

- **create table** *course* (

  …
  dept_name **varchar**(20),
  foreign key (*dept_name*) **references** *department*
  on delete cascade
  on update cascade,
  . . .
)

- alternative actions to cascade: **set null**, **set default**
- First, we have to insert data in the department table and then in the course table

Integrity Constraint Violation During Transactions

- E.g.

  create table *person* (
  ID **char**(10),
  name **char**(40),
  mother **char**(10),
  father **char**(10),
  primary key *ID,*
  foreign key *father* **references** *person,*
  foreign key *mother* **references** *person*)

- How to insert a tuple without causing constraint violation ?
  - insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking

Integrity Constraint Violation During Transactions

- Basic operations that change the states of relations in the database:
  - Insert.
  - Delete.
  - Update
- Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated.

Integrity Constraint Violation During Transactions

Example: **insert** operation

- Provides a list of attribute values for a new tuple that is to be inserted into a relation R
- Can violate any of the types of constraints
  - Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain (dayOfWeek=January) or is not of the appropriate data type (dayOfWeek=3.5).
  - Key constraints can be violated if a key value in the new tuple already exists in another tuple in the relation.
  - Entity integrity can be violated if any part of the primary key of the new tuple is NULL .
  - Referential integrity can be violated if the value of any foreign key in the new tuple refers to a tuple that does not exist in the referenced relation.
- If an insertion violates one or more constraints
  - Default option is to reject the insertion

Integrity Constraint Violation During Transactions

Example: **delete** operation

- Can violate only referential integrity
- If tuple being deleted is referenced by foreign keys from other tuples:
  - Restrict: Reject the deletion
  - Cascade: Propagate the deletion by deleting tuples that reference the tuple that is being deleted
  - Set null or set default: Modify the referencing attribute values that cause the violation

Integrity Constraint Violation During Transactions

Example: **update** operation

- It is necessary to specify a condition on attributes of a relation
  - Select the tuple (or tuples) to be modified
- If the attribute/s not part of a primary key nor of a foreign key
  - Usually causes no problems
- Updating a primary/foreign key
  - Similar issues as with Insert/Delete

Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema

- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list>

  on <relation name or view name> **to** <user list>

- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

    grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list>

  on <relation name or view name> **from** <user list>

- Example:

  revoke select on *branch* **from** U_1, U_2, U_3

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

Roles

- **create role** secretary;
- **grant** *secretary* **to James;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *secretary*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *secretary*;
    - *Secretary* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *teaching_assistant* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mary;
  - Mary is able to define FKs associated to the attribute dept_name of table department when creating relations.
- transfer of privileges
  - **grant select on** *department* **to** James **with grant option**;
    - It means that James has the select privilege on department and can grant the select privilege to other users/roles
  - **revoke select on** *department* **from** James, Satoshi **cascade**;
    - James, Satoshi (and other users/roles) loose the select privilege on table department
  - **revoke select on** *department* **from** James, Satoshi **restrict**;
    - An error is returned if there is a revocation of privileges in chain. In that case, the revocation of privileges is not taken into account.

Triggers

- For semantic integrity constraints we user triggers
- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - **E.g., after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** **:** for deletes and updates
  - **referencing new row as** **:** for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blank grades to null.

```
    create trigger setnull_trigger before update of takes
    referencing new row as nrow
    for each row
    when (nrow.grade = ‘ ‘)
begin atomic
        set nrow.grade = null;
end;
```

Trigger Example

- **Example**: In a bank when the balance of an account is negative. The bank can allow you to have negative balances by creating debt loans. The actions would be: insert a row in loan with the negative amount, insert a new row in borrower and make the balance of the account equal to zero.

```
create trigger manageDebts after update on account
    referencing new row as nrow
    for each row
    when (nrow.balance < 0
    begin atomic
        insert into borrower

                (select customerName, accountNumber from depositor where
                nrow.accountNumber=depositor.accountNumber);

        insert into loan values (nrow.accountNumber, nrow.branchName, -nrow.balance)

         update account set balance=0

            where account.accountNumber=nrow.accountNumber
    end;
```

Trigger Example

- **Example**: In a warehouse when the stock drops a certain amount. When modifying the inventory level of a product, the trigger must compare the level with the minimum value allowed and if the level is below the minimum, a new order is added to the Orders relationship.

```
create trigger newOrder after update of amount on inventory
referencing old row as orow, new row as nrow
for each row
when (nrow.level < (select level from minimumLevel
                    where minimumLevel.product = orow.product)
and orow.level > (select level from minimumLevel
                    where minimumLevel.product = orow.product)
begin atomic
   insert into Orders (
         select product, amount from newOrder where newOrder.product = orow.product)
end;
```

Trigger Example

Example: the trigger adds the credits of a course to the attribute tot_cred if the student has passed the course (when the new value of grade is not 'F' and is not null, and the old value of grade was 'F' or null). Thus, the course was failed but now it has been passed.

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
   and (orow.grade = 'F' or orow.grade is null)
begin atomic
    update student
    set tot_cred= tot_cred +
         (select credits
          from course
          where course.course_id= nrow.course_id)
    where student.id = nrow.id;
end;
```

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication

Functional Dependencies

- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key.*
- *Examples:*

  SSN→ Name

  ProjectNumber → ProjectName, ProjectLocalization

  SSN, ProjectNumber → NumberOfHours

Functional Dependencies (Cont.)

- Let *R* be a relation schema

  $\alpha \subseteq R$ and $\beta \subseteq R$
- The **functional dependency**

  $\alpha \rightarrow \beta$

  holds on *R* if and only if for any legal relations *r*(R), whenever any two tuples t_1 and t_2 of *r* agree on the attributes α, they also agree on the attributes β. That is,

  $t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$
- Example: Consider *r*(A,*B*) with the following instance of *r.*

1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.
- A→B holds if a value in A is not related to several values in B.
- There should not be 2 rows with the same value in A and different values in B (The same NID cannot belong to two different people)

Functional Dependencies (Cont.)

Exercise: Obtain functional dependencies:

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Functional Dependencies (Cont.)

Exercise: Obtain functional dependencies:

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

$A \rightarrow C$ holds (Each row in A implies one row in C)
$C \rightarrow A$ holds (Each row in C involves more than one row in A
$AB \rightarrow D$ holds (all values in AB are different)
Trivial: $A \rightarrow A$
Trivial dependence $\alpha \rightarrow \beta$, if $\beta \subseteq \alpha$
Example $AB \rightarrow A$

Functional Dependencies (Cont.)

- *K* is a superkey for relation schema *R* if and only if $K \rightarrow R$
  - *Example: SSN → name, address, telephone.*

- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

  inst_dept (*<u>ID,</u> name, salary<u>, dept_name,</u> building, budget*).

  We expect these functional dependencies to hold:

  $dept\_name \rightarrow building$

  and $ID \rightarrow building$

  but would not expect the following to hold:

  $dept\_name \rightarrow salary$

Use of Functional Dependencies

- We use functional dependencies to:
  - test relations to see if they are legal under a given set of functional dependencies.
    - If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* **satisfies** *F.*
  - specify constraints on the set of legal relations
    - We say that *F* **holds on** *R* if all legal relations on *R* satisfy the set of functional dependencies *F.*

Functional Dependencies (Cont.)

- *A* functional dependency is **trivial** if it is satisfied by all instances of a relation
  - Example*:*
    - *ID, name* $\rightarrow$ *ID*
    - *name* $\rightarrow$ *name*
  - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Closure of a Set of Functional Dependencies

- Given a set *F* of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.
  - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by *F* is the **closure** of *F*.
- We denote the *closure* of *F* by $\mathbf{F^+}$
- F^+ is a superset of *F*.

Closure of a Set of Functional Dependencies

- We can find F^{+}, the closure of F, by repeatedly applying **Armstrong's Axioms:**
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (**reflexivity**)
  - if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ (**augmentation**)
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (**transitivity**)
- These rules are
  - **sound** (generate only functional dependencies that actually hold), and
  - **complete** (generate all functional dependencies that hold).

Example

- $R = (A, B, C, G, H, I)$
  $F = \{ A \to B$
  $A \to C$
  $CG \to H$
  $CG \to I$
  $B \to H\}$
- some members of F^+
  - $A \to H$
    - by transitivity from $A \to B$ *and* $B \to H$
  - $AG \to I$
    - by augmenting $A \to C$ with G, to get $AG \to CG$ and then transitivity with $CG \to I$
  - $CG \to HI$
    - by augmenting $CG \to I$ to infer $CG \to CGI$, and augmenting of $CG \to H$ to infer $CGI \to HI$, and then transitivity

Procedure for Computing F^+

- To compute the closure of a set of functional dependencies F:

```
$F^+ = F$
repeat
    for each functional dependency $f$ in $F^+$
        apply reflexivity and augmentation rules on $f$
        add the resulting functional dependencies to $F^+$
    for each pair of functional dependencies $f_1$ and $f_2$ in $F^+$
        if $f_1$ and $f_2$ can be combined using transitivity
         then add the resulting functional dependency to $F^+$
until $F^+$ does not change any further
```

NOTE: We shall see an alternative procedure for this task later

Closure of Functional Dependencies (Cont.)

- Additional rules:
  - If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (**union**)
  - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (**decomposition**)
  - If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds (**pseudotransitivity**)

  The above rules can be inferred from Armstrong's axioms.

Closure of Attribute Sets

- Given a set of attributes α, define the ***closure*** of α **under** *F* (denoted by α^+) as the set of attributes that are functionally determined by α under *F*

- Algorithm to compute α^+, the closure of α under *F*

```
result := $\alpha$;
while (changes to result) do
    for each $\beta \rightarrow \gamma$ in F do
        begin
            if $\beta \subseteq$ result then  result := result $\cup \gamma$
        end
```

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- $(AG)^+$
  1. *result* = *AG*
  2. *result* = *ABCG* $(A \rightarrow C$ and $A \rightarrow B$ and $A \subseteq AG)$
  3. *result* = *ABCGH* $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. *result* = *ABCGHI* $(CG \rightarrow I$ and $CG \subseteq AGBCH)$

- Is AG a super key? Yes, it is
  - Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq$ R

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
  - To test if α is a superkey, we compute α^{+}, and check if α^{+} contains all attributes of *R*.
- Testing functional dependencies
  - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^{+}), just check if $\beta \subseteq \alpha^{+}$.
  - That is, we compute α^{+} by using attribute closure, and then check if it contains β.
  - Is a simple and cheap test, and very useful
- Computing closure of F
  - For each $\gamma \subseteq R$, we find the closure γ^{+}, and for each $S \subseteq \gamma^{+}$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - E.g.: on RHS: $\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow CD\}$ can be simplified to
      $$\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow D\}$$
    - E.g.: on LHS: $\{A \rightarrow B,\ B \rightarrow C,\ AC \rightarrow D\}$ can be simplified to
      $$\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow D\}$$
- Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

Extraneous/Reduntant Attributes

- Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.
  - Attribute A is **extraneous** in α if $A \in \alpha$ and *F* logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - Attribute *A* is **extraneous** in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies *F.*
- Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$
  - *B* is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (I.e. the result of dropping *B* from $AB \rightarrow C$).
- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
  - *C* is extraneous in $AB \rightarrow CD$ since $AB \rightarrow C$ can be inferred even after deleting *C*

Testing if an Attribute is Extraneous

- Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.
- To test if attribute A $\in \alpha$ is extraneous in α
  1. compute $(\{\alpha\} - A)^+$ using the dependencies in *F*
  2. check that $(\{\alpha\} - A)^+$ contains β; if it does, *A* is extraneous in α
- To test if attribute *A* $\in \beta$ is extraneous in β
  1. compute α^+ using only the dependencies in
     $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,
  2. check that α^+ contains *A;* if it does, *A* is extraneous in β

Canonical Cover

- A **canonical cover** for *F* is a set of dependencies F_c such that
  - *F* logically implies all dependencies in $F_{c,}$ and
  - F_c logically implies all dependencies in *F,* and
  - No functional dependency in F_c contains an extraneous attribute, and
  - Each left side of functional dependency in F_c is unique.
- To compute a canonical cover for *F*:

```
repeat
    Use the union rule to replace any dependencies in $F$
        $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$
    Find a functional dependency $\alpha \rightarrow \beta$ with an
        extraneous attribute either in $\alpha$ or in $\beta$
            /* Note: test for extraneous attributes done using $F_{c,}$ not F*/
    If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
until $F$ does not change
```

- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

Computing a Canonical Cover

- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
  - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
    - Yes: in fact, $B \rightarrow C$ is already present!
  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
    - Yes: using transitivity on $A \rightarrow B$ *and* $B \rightarrow$ C.
      - Can use attribute closure of A in more complex cases
- The canonical cover is: $A \rightarrow B$
  $B \rightarrow C$