Relational Databases

Integrity Constraints:

Relational:

- A relation is a named, two-dimensional table of data
- Every relation has a unique name, and consists of a set of named columns and an arbitrary number of unnamed rows
- An attribute is a named column of a relation, and every attribute value is atomic.
- Every row is unique, and corresponds to a record that contains data attributes for a single entity.
- The order of the columns is irrelevant.
- The order of the rows is irrelevant.

Relational Structure:

- We can express the structure of a relation by a Tuple, a shorthand notation
- The name of the relation is followed (in parentheses) by the names of the attributes of that relation, e.g.:
- EMPLOYEE1(Emp_ID, Name, Dept, Salary)

Relational Keys:

- Must be able to store and retrieve a row of data in a relation, based on the data values stored in that row
- A primary key is an attribute (or combination of attributes) that uniquely identifies each row in a relation.
- The primary key in the EMPLOYEE1 relation is EMP_ID (this is why it is underlined) as in:
- EMPLOYEE1(Emp_ID, Name, Dept, Salary)

Composite and Foreign Keys:

- A Composite key is a primary key that consists of more than one attribute.
- e.g., the primary key for the relation DEPENDENT would probably consist of the combination Emp-ID and Dependent_Name
- A Foreign key is used when we must represent the relationship between two tables and relations
- A foreign key is an attribute (possibly composite) in a relation of a database that serves as the primary key of another relation in the same database

Foreign Keys:

- Consider the following relations:
- EMPLOYEE1(Emp_ID, Name, Dept_Name,Salary)
- DEPARTMENT(Dept_Name, Location, Fax)
- The attribute Dept_Name is a foreign key in EMPLOYEE1. It allows the user to associate any employee wit the department they are assigned to.
- Some authors show the fact that an attribute is a foreign key by using a dashed underline.

Integrity Constraints:

- These help maintain the accuracy and integrity of the data in the database
- Domain Constraints - a domain is the set of allowable values for an attribute.
- Domain definition usually consists of 4 components: domain name, meaning, data type, size (or length), allowable values/allowable range (if applicable)
- Entity Integrity ensures that every relation has a primary key, and that all the data values for that primary key are valid. No primary key attribute may be null.

Entity Integrity:

- In some cases a particular attribute cannot be assigned a data value, e.g. when there is no applicable data value or the value is not known when other values are assigned
- In these situations we can assign a null value to an attribute (null signifies absence of a value)
- But still primary key values cannot be null – the entity integrity rule states that "no primary key attribute (or component of a primary key attribute) may be null

Referential Integrity:

- A Referential Integrity constraint is a rule that maintains consistency among the rows of two relations – it states that any foreign key value (on the relation of the many side) MUST match a primary key value in the relation of the one side. (Or the foreign key can be null)
- In the following Fig., an arrow has been drawn from each foreign key to its associated primary key. A referential integrity constraint must be defined for each of these arrows in the schema
- How do you know if a foreign key is allowed to be null?
- In this example, as each ORDER must have a CUSTOMER the foreign key of Customer_ID cannot be null on the ORDER relation
- Whether a foreign key can be null must be specified as a property of the foreign key attribute when the database is designed
- Whether foreign key can be null can be complex to model, e.g. what happens to order data if we choose to delete a customer who has submitted orders? We may want to see sales even though we do not care about the customer anymore. 3 choices are possible:
- Restrict – don't allow delete of "parent" side if related rows exist in "dependent" side, i.e. prohibit deletion of the customer until all associated orders are first deleted

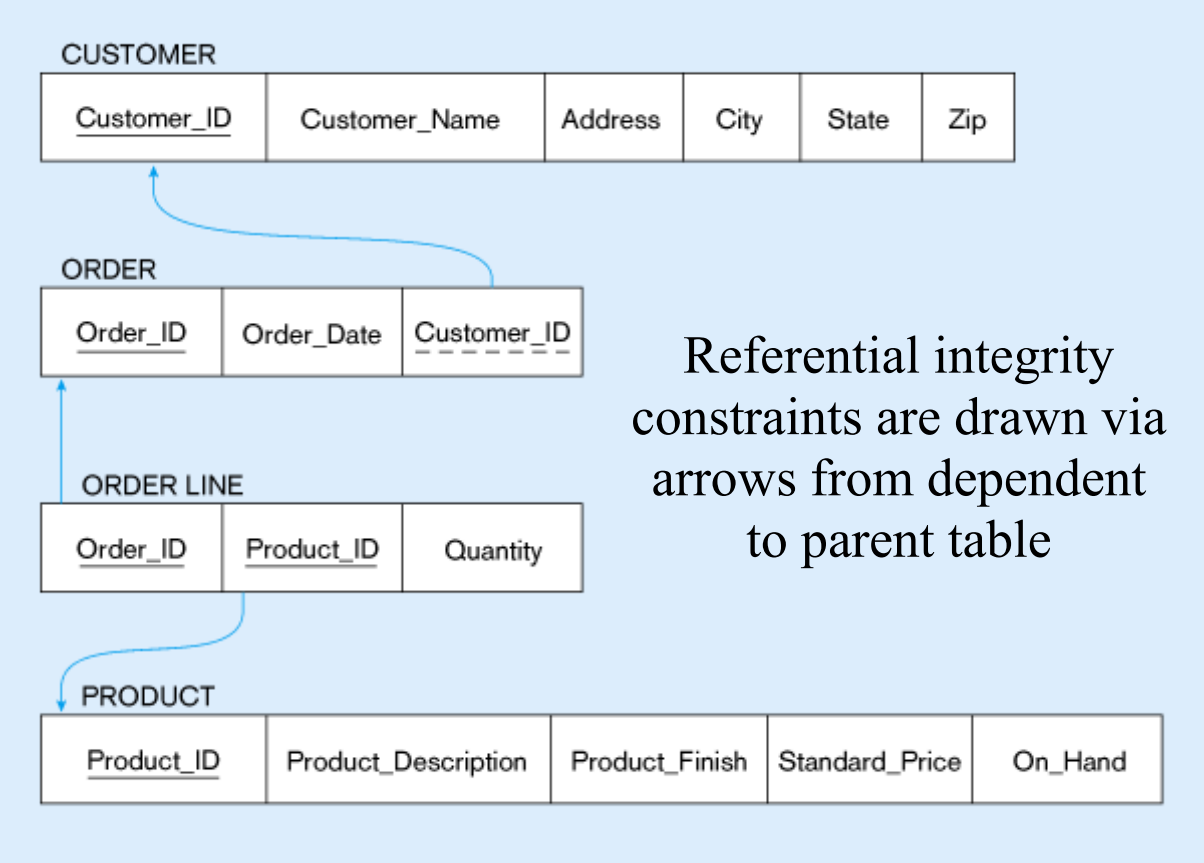
Solution:

Cascade – automatically delete "dependent" side rows that correspond with the "parent" side row to be deleted, i.e. delete the associated orders, in which case we lose not only the customer but also the sales history

Set-to-Null – set the foreign key in the dependent side to null if deleting from the parent side - an exception that says although an order must have a customer_ID value when the order is created, Customer_ID can become null later if the associated customer is deleted [not allowed for weak entities]

Example:

Referential integrity constraints (Pine Valley Furniture)



By

Prof. Ramesh D Hari Nandan
Faculty of Computer Science and Engineering
Indian Institute of Technology (Indian School of Mines)
drramesh@iitism.ac.in

Relational Database Design

- Pitfalls in Relational Database Design
- Functional Dependencies
- Multivalued Dependencies and Fourth Normal Form

Pitfalls in Relational Database Design

- Relational database design requires that we find a “good” collection of relation schemas. A bad design may lead to
  - Repetition of Information.
  - Inability to represent certain information.
- Design Goals:
  - Avoid redundant data
  - Ensure that relationships among attributes are represented
  - Facilitate the checking of updates for violation of database integrity constraints.

Example

- Consider the relation schema:

  Lending-schema = (*branch-name, branch-city, assets, customer-name, loan-number, amount)*

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500

- Redundancy:
  - Data for *branch-name, branch-city,* assets are repeated for each loan that a branch makes
  - Wastes space
  - Complicates updating, introducing possibility of inconsistency of *assets* value
- Null values
  - Cannot store information about a branch if no loans exist
  - Can use null values, but they are difficult to handle.

Decomposition

- Decompose the relation schema *Lending-schema* into:

Branch-schema = (branch-name, branch-city,assets)

Loan-info-schema = (customer-name, loan-number, branch-name, amount)

- All attributes of an original schema (*R)* must appear in the decomposition (R_1, R_2)*:*

$$R = R_1 \cup R_2$$

- Lossless-join decomposition.
  For all possible relations *r* on schema *R*

$$r = \Pi_{R1}(r) \quad \Pi_{R2}(r)$$

$$\bowtie$$

Example of Non Lossless-Join Decomposition

- Decomposition of $R = (A, B)$

  $R_1 = (A)$ $R_2 = (B)$

A	B
α	1
α	2
β	1

r

A
α
β

$\Pi_A(r)$

B
1
2

$\Pi_{B(r)}$

$\Pi_A (r) \bowtie \Pi_B (r)$

A	B
α	1
α	2
β	1
β	2

Goal — Devise a Theory for the Following

- Decide whether a particular relation *R* is in “good” form.
- In the case that a relation *R* is not in “good” form, decompose it into a set of relations {R_1, R_2, ..., R_n} such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on:
  - functional dependencies
  - multivalued dependencies

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.*

Functional Dependencies (Cont.)

- Let *R* be a relation schema

$$\alpha \subseteq R \quad and \quad \beta \subseteq R$$

- 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,

$$t_1[\alpha] = t_2[\alpha] \quad \Rightarrow \quad t_1[\beta] = t_2[\beta]$$

- Example: Consider *r(A,B)* with the following instance of *r.*

- On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

1	4
1	5
3	7

Functional Dependencies (Cont.)

- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
  - $K \rightarrow R$, and
  - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

  Loan-info-schema = (customer-name, loan-number, branch-name, amount).

  We expect this set of functional dependencies to hold:

  loan-number $\rightarrow$ *amount*
  loan-number $\rightarrow$ *branch-name*

  but would not expect the following to hold:

  loan-number $\rightarrow$ *customer-name*

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.*
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
  - For example, a specific instance of *Loan-schema* may, by chance, satisfy
    loan-number $\rightarrow$ *customer-name.*

Functional Dependencies (Cont.)

- *A* functional dependency is trivial if it is satisfied by all instances of a relation
  - *E.g.*
    - *customer-name, loan-number* $\rightarrow$ *customer-name*
    - *customer-name* $\rightarrow$ *customer-name*
  - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Closure of a Set of Functional Dependencies

- Given a set *F* set of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.
  - E.g. 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 F^+.
- We can find all of F^+ by 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

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - from $CG \rightarrow H$ *and* $CG \rightarrow I$: "union rule" can be inferred from
      - definition of functional dependencies, or
      - Augmentation of $CG \rightarrow I$ to infer $CG \rightarrow CGI$, augmentation of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, 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 will see an alternative procedure for this task later

Closure of Functional Dependencies (Cont.)

- We can further simplify manual computation of F^+ by using the following 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:

  $\alpha \rightarrow \beta$ is in $F^+ \Leftrightarrow \beta \subseteq \alpha^+$

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

```
result := α;
while (changes to result) do
    for each β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ
        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 $A \rightarrow B)$
  3. $result = ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$
- Is AG a candidate key?
  1. Is AG a super key?
     1. Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq R$
  2. Is any subset of AG a superkey?
     1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$
     2. Does $G \rightarrow R$? == Is $(G)^+ \supseteq R$

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
  - Eg: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - E.g. on RHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow CD\}$ can be simplified to $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$
    - E.g. on LHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad AC \rightarrow D\}$ can be simplified to $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$
- 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

Design Goals

- Goal for a relational database design is:
  - BCNF.
  - Lossless join.
  - Dependency preservation.
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation
  - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.

  Can specify FDs using assertions, but they are expensive to test
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

Testing for FDs Across Relations

- If decomposition is not dependency preserving, we can have an extra **materialized view** for each dependency $\alpha \rightarrow \beta$ in F_c that is not preserved in the decomposition
- The materialized view is defined as a projection on $\alpha\ \beta$ of the join of the relations in the decomposition
- Many newer database systems support materialized views and database system maintains the view when the relations are updated.
  - No extra coding effort for programmer.
- The functional dependency $\alpha \rightarrow \beta$ is expressed by declaring α as a candidate key on the materialized view.
- Checking for candidate key cheaper than checking $\alpha \rightarrow \beta$
- BUT:
  - Space overhead: for storing the materialized view
  - Time overhead: Need to keep materialized view up to date when relations are updated
  - Database system may not support key declarations on materialized views

Multivalued Dependencies

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a database

  classes(course, teacher, book)

  such that (*c,t,b*) $\in$ *classes* means that *t* is qualified to teach *c,* and *b* is a required textbook for *c*
- The database is supposed to list for each course the set of teachers any one of which can be the course's instructor, and the set of books, all of which are required for the course (no matter who teaches it).

Multivalued Dependencies (Cont.)

course	*teacher*	*book*
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Shaw
operating systems	Jim	OS Concepts
operating systems	Jim	Shaw

classes

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if Sara is a new teacher that can teach database, two tuples need to be inserted

(database, Sara, DB Concepts)
(database, Sara, Ullman)

Multivalued Dependencies (Cont.)

- Therefore, it is better to decompose *classes* into:

course	*teacher*
database	Avi
database	Hank
database	Sudarshan
operating systems	Avi
operating systems	Jim

teaches

course	*book*
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

text

We shall see that these two relations are in Fourth Normal Form (4NF)

Multivalued Dependencies (MVDs)

- Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The *multivalued dependency*

$$\alpha \twoheadrightarrow \beta$$

holds on R if in any legal relation $r(R)$, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$\begin{aligned} t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\ t_3[\beta] &= t_1[\beta] \\ t_3[R - \beta] &= t_2[R - \beta] \\ t_4[\beta] &= t_2[\beta] \\ t_4[R - \beta] &= t_1[R - \beta] \end{aligned}$$

MVD (Cont.)

- Tabular representation of $\alpha \twoheadrightarrow \beta$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

Example

- Let *R* be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.

  Y, Z, W

- We say that $Y \rightarrow\rightarrow Z$ (*Y* multidetermines *Z*) if and only if for all possible relations *r(R)*

  $< y_1, z_1, w_1 > \in r$ and $< y_2, z_2, w_2 > \in r$

  then

  $< y_1, z_1, w_2 > \in r$ and $< y_2, z_2, w_1 > \in r$

- Note that since the behavior of *Z* and *W* are identical it follows that $Y \rightarrow\rightarrow Z$ if $Y \rightarrow\rightarrow W$

Example (Cont.)

- In our example:

  $course \rightarrow\!\!\!\!\rightarrow$ teacher
  $course \rightarrow\!\!\!\!\rightarrow$ *book*

- The above formal definition is supposed to formalize the notion that given a particular value of *Y* (*course*) it has associated with it a set of values of *Z (teacher)* and a set of values of *W (book)*, and these two sets are in some sense independent of each other.
- Note:
  - If $Y \rightarrow Z$ then $Y \rightarrow\!\!\!\!\rightarrow Z$
  - Indeed we have (in above notation) $Z_1 = Z_2$
    The claim follows.

Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:
  1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies
  2. To specify constraints on the set of legal relations. We shall thus concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.
- If a relation *r* fails to satisfy a given multivalued dependency, we can construct a relations *r′* that does satisfy the multivalued dependency by adding tuples to *r.*

Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:
  - If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$

  That is, every functional dependency is also a multivalued dependency
- The **closure** D^+ of *D* is the set of all functional and multivalued dependencies logically implied by *D*.
  - We can compute D^+ from *D*, using the formal definitions of functional dependencies and multivalued dependencies.
  - We can manage with such reasoning for very simple multivalued dependencies, which seem to be most common in practice
  - For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules (see Appendix C).

Fourth Normal Form

- A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\!\!\!\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \rightarrow\!\!\!\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
  - α is a superkey for schema R
- If a relation is in 4NF it is in BCNF

Restriction of Multivalued Dependencies

- The restriction of D to R_i is the set D_i consisting of
  - All functional dependencies in D^+ that include only attributes of R_i
  - All multivalued dependencies of the form

    $\alpha \rightarrow\rightarrow (\beta \cap R_i)$

    where $\alpha \subseteq R_i$ and $\alpha \rightarrow\rightarrow \beta$ is in D^+