Relational Database Design

Based on slides from the text book: Database System Concepts, 5th Ed.

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
- Functional Dependency Theory
- Algorithms for Functional Dependencies
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data
The Banking Schema

- branch = \((\text{branch\_name, branch\_city, assets})\)
- customer = \((\text{customer\_id, customer\_name, customer\_street, customer\_city})\)
- loan = \((\text{loan\_number, amount})\)
- account = \((\text{account\_number, balance})\)
- employee = \((\text{employee\_id, employee\_name, telephone\_number, start\_date})\)
- dependent\_name = \((\text{employee\_id, dname})\)
- account\_branch = \((\text{account\_number, branch\_name})\)
- loan\_branch = \((\text{loan\_number, branch\_name})\)
- borrower = \((\text{customer\_id, loan\_number})\)
- depositor = \((\text{customer\_id, account\_number})\)
- cust\_banker = \((\text{customer\_id, employee\_id, type})\)
- works\_for = \((\text{worker\_employee\_id, manager\_employee\_id})\)
- payment = \((\text{loan\_number, payment\_number, payment\_date, payment\_amount})\)
- savings\_account = \((\text{account\_number, interest\_rate})\)
- checking\_account = \((\text{account\_number, overdraft\_amount})\)

Combine Schemas?

- Suppose we combine borrower and loan to get
  \(\text{bor\_loan} = ((\text{customer\_id, loan\_number, amount}))\)
- Result is possible repetition of information (L-100 in example below)
A Combined Schema Without Repetition

- Consider combining loan_branch and loan

  \[ \text{loan_amt_br} = (\text{loan_number}, \text{amount}, \text{branch_name}) \]

- No repetition (as suggested by example below)

What About Smaller Schemas?

- Suppose we had started with bor_loan. How would we know to split up (decompose) it into borrower and loan?

- Write a rule "if there were a schema \((\text{loan_number}, \text{amount})\), then \(\text{loan_number}\) would be a candidate key"

- Denote as a functional dependency:

  \[ \text{loan_number} \rightarrow \text{amount} \]

- In bor_loan, because \(\text{loan_number}\) is not a candidate key, the amount of a loan may have to be repeated. This indicates the need to decompose bor_loan.

- Not all decompositions are good. Suppose we decompose employee into

  \begin{align*}
  \text{employee1} &= (\text{employee_id}, \text{employee_name}) \\
  \text{employee2} &= (\text{employee_name}, \text{telephone_number}, \text{start_date})
  \end{align*}

- The next slide shows how we lose information -- we cannot reconstruct the original employee relation -- and so, this is a lossy decomposition.
A Lossy Decomposition

First Normal Form

- Domain is atomic if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in first normal form if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - Example: Set of accounts stored with each customer, and set of owners stored with each account
  - We assume all relations are in first normal form
First Normal Form (Cont’d)

- Atomicity is actually a property of how the elements of the domain are used.
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form CS0012 or EE1127
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

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
Normalization

- Normalization is a design technique that is widely used as a guide in designing relational databases. A two step process
  - that puts data into tabular form by removing repeating groups
  - removes duplicated data from the relational tables.
- Based on the concepts of normal forms. A relational table is in a particular normal form if it satisfies a certain set of constraints.
- Five normal forms that have been defined (manily by E.F. Codd)

Basic Concepts
- Goal of normalization: create a set of relational tables that
  - are free of redundant data
  - can be consistently and correctly modified.
- All tables in a relational database should be in the third normal form (3NF).
- A relational table is in 3NF if and only if all non-key columns are
  - mutually independent: no non-key column is dependent upon any combination of the other columns.
  - fully dependent upon the primary key.
  - The first two normal forms are intermediate steps to achieve the goal of having all tables in 3NF.
- Key preliminary concepts:
  - functional dependencies
  - lossless decomposition
**Functional Dependencies**

- The concept of functional dependencies is the basis for the first three normal forms.
- A column, \( Y \), of the relational table \( R \) is said to be **functionally dependent** upon column \( X \) of \( R \) if and only if
  - each value of \( X \) in \( R \) is associated with precisely one value of \( Y \) at any given time.
  - \( X \) and \( Y \) may be composite.
- Saying that column \( Y \) is functionally dependent upon \( X \) is the same as saying the values of column \( X \) identify the values of column \( Y \). If column \( X \) is a primary key, then all columns in the relational table \( R \) must be functionally dependent upon \( X \). Constraints on the set of legal relations.
- A short-hand notation for describing a functional dependency is:
  \[ R.x \rightarrow R.y \]
- In the relational table named \( R \), column \( x \) functionally determines (identifies) column \( y \)

---

**Functional Dependencies (Cont.)**

- Let \( R \) be a relation schema
  \[ \alpha \subseteq R \text{ and } \beta \subseteq R \]
- 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 \( \alpha \), they also agree on the attributes \( \beta \). 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 \( r \).

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

On this instance, \( A \rightarrow B \) does **NOT** hold, but \( B \rightarrow A \) does hold.
Functional Dependencies (Cont.)

- K is a superkey for relation schema R if and only if K → R
- K is a candidate key for R if and only if
  - K → R, and
  - for no α ⊂ K, α → R

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

bor_loan = (customer_id, loan_number, amount).

We expect this functional dependency to hold:

loan_number → amount

but would not expect the following to hold:

amount → 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 may, by chance, satisfy amount → customer_name.
Functional Dependencies (Cont.)

- A functional dependency is trivial if it is satisfied by all instances of a relation
  - Example:
    - customer\_name, loan\_number → customer\_name
    - customer\_name → customer\_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 \( F^+ \).
- \( F^+ \) is a superset of \( F \).
First Normal Form (1NF)

FIRST (s#, status, city, p#, qty)

- s# supplier identification number - this is the primary key
- status status code assigned to city
- cityname of city where supplier is located
- p# part number of part supplied
- qty> quantity of parts supplied to date

Example inspired by:
“Introduction to Data Modeling”
(Univ. of Texas © Austin)
http://www.utexas.edu/its/archive/windows/database/datamodeling/index.html

Problems with 1NF

- FIRST contains redundant data.
  - supplier's location and the location's status have to be repeated for every part supplied.
- Redundancy causes what are called update anomalies. Update anomalies are problems that arise when information is inserted, deleted, or updated.
- The following anomalies could occur in FIRST:
  - INSERT. supplier (s5) is located in a particular city (Athens) cannot be added until they supplied a part.
  - DELETE. If a row is deleted, then also information about the supplier is lost.
  - UPDATE. If supplier s1 moved from London to New York, then six rows would have to be updated with this new information.
Second Normal Form (2NF)

- A relational table is in second normal form 2NF if
  - it is in 1NF and
  - every non-key column is fully dependent upon the primary key. That is, every non-key column must be dependent upon the entire primary key.

- FIRST is in 1NF but not in 2NF because status and city are functionally dependent upon only on the column s# of the composite key (s#, p#). This can be illustrated by listing the functional dependencies in the table:
  - s# → city
  - status city → status
  - (s#, p#) → qty

<table>
<thead>
<tr>
<th></th>
<th>s#</th>
<th>city</th>
<th>p#</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>London</td>
<td>p1</td>
<td>800</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>London</td>
<td>p2</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>London</td>
<td>p3</td>
<td>400</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>London</td>
<td>p4</td>
<td>200</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>London</td>
<td>p5</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>London</td>
<td>p6</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Paris</td>
<td>p1</td>
<td>300</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>Paris</td>
<td>p2</td>
<td>200</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>London</td>
<td>p2</td>
<td>200</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>London</td>
<td>p4</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>London</td>
<td>p5</td>
<td>400</td>
</tr>
</tbody>
</table>

2NF - example

- The process for transforming a 1NF table to 2NF is:
  - identify any attributes, not in composite key, and the columns they determine.
  - Create new table for each such attribute and the unique columns it determines.
  - Move determined columns to the new table. Determinant attr. becomes primary key.
  - Delete the columns moved from the original table except for the determinant: serves as a foreign key.

<table>
<thead>
<tr>
<th></th>
<th>s#</th>
<th>city</th>
<th>p#</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>London</td>
<td>p1</td>
<td>300</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>London</td>
<td>p2</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>London</td>
<td>p3</td>
<td>400</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>London</td>
<td>p4</td>
<td>200</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>London</td>
<td>p5</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>London</td>
<td>p6</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Paris</td>
<td>p1</td>
<td>300</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>Paris</td>
<td>p2</td>
<td>200</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>London</td>
<td>p2</td>
<td>200</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>London</td>
<td>p4</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>London</td>
<td>p5</td>
<td>400</td>
</tr>
</tbody>
</table>

Tables in 2NF still suffer modification anomalies:
In SECOND, they are:
- INSERT. The fact that a particular city has a certain status (Rome has a status of 50) cannot be inserted until there is a supplier in the city.
- DELETE. Deleting any row in SUPPLIER destroys the status information about the city as well as the association between supplier and city.
Third Normal Form

- The third normal form requires that all non-key attributes are functionally dependent only upon the primary key – with NO transitive dependencies.

- A transitive dependency occurs when:
  - a non-key column that is a determinant of the primary key is the determinate of other columns.

- A relation schema \( R \) is in third normal form (3NF) if for all:
  \[ \alpha \rightarrow \beta \text{ in } F^+ \]
  at least one of the following holds:
  - \( \alpha \rightarrow \beta \) is trivial (i.e., \( \beta \in \alpha \))
  - \( \alpha \) is a superkey for \( R \)
  - Each attribute \( A \) in \( \beta - \alpha \) is contained in a candidate key for \( R \).
    (NOTE: each attribute may be in a different candidate key)

- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).

- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later). 

Table PARTS is already in 3NF. The non-key column, \( qty \), is fully dependent upon the primary key (\( s# \), \( p# \)).

SUPPLIER is in 2NF but not in 3NF because it contains a transitive dependency: non-key column that is a determinant of the primary key is the determinate of other columns.

The concept of a transitive dependency can be illustrated by showing the functional dependencies in SUPPLIER:

- SUPPLIER.s# \( \rightarrow \) SUPPLIER.status
- SUPPLIER.s# \( \rightarrow \) SUPPLIER.city
- SUPPLIER.city \( \rightarrow \) SUPPLIER.status
Third Normal form

- The process of transforming a table into 3NF:
  - Identify any determinants, other the primary key, and the columns they determine.
  - Create and name a new table for each determinant and the unique columns it determines.
  - Move the determined columns from the original table to the new table. The determinate becomes the primary key of the new table.
  - Delete the columns you just moved from the original table except for the determinate which will serve as a foreign key.
  - The original table may be renamed to maintain semantic meaning.

- SUPPLIER into 3NF
  - new table called CITY_STATUS
  - move the columns city and status into it.
  - Status is deleted from the original table,
  - city is left behind to serve as a foreign key to CITY_STATUS.
  - original renamed to SUPPLIER_CITY to reflect its semantics

<table>
<thead>
<tr>
<th>SUPPLIER_CITY</th>
<th>CITY_STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>city</td>
</tr>
<tr>
<td>s1</td>
<td>London</td>
</tr>
<tr>
<td>s2</td>
<td>Paris</td>
</tr>
<tr>
<td>s3</td>
<td>Paris</td>
</tr>
<tr>
<td>s4</td>
<td>London</td>
</tr>
<tr>
<td>s5</td>
<td>Athens</td>
</tr>
</tbody>
</table>

Advantages of Third Normal Form

- The advantage of having relational tables in 3NF is that it eliminates redundant data which in turn saves space and reduces manipulation anomalies. For example, the improvements to our sample database are:

- INSERT. Facts about the status of a city, Rome has a status of 50, can be added even though there is not supplier in that city. Likewise, facts about new suppliers can be added even though they have not yet supplied parts.

- DELETE. Information about parts supplied can be deleted without destroying information about a supplier or a city. UPDATE. Changing the location of a supplier or the status of a city requires modifying only one row.

<table>
<thead>
<tr>
<th>SUPPLIER_CITY</th>
<th>CITY_STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>London</td>
</tr>
<tr>
<td>s2</td>
<td>Paris</td>
</tr>
<tr>
<td>s3</td>
<td>Paris</td>
</tr>
<tr>
<td>s4</td>
<td>London</td>
</tr>
<tr>
<td>s5</td>
<td>Athens</td>
</tr>
</tbody>
</table>
Boyce-Codd Normal Form

A relation schema $R$ is in BCNF with respect to a set $F$ of functional dependencies if for all functional dependencies in $F$ of the form 
\[ \alpha \rightarrow \beta \]
where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- $\alpha$ is a superkey for $R$

A relational table is in BCNF if and only if every determinant is a candidate key.

Difference to 3NF: it is not necessary for the property to hold:

Each attribute $A$ in $\beta - \alpha$ is contained in a candidate key for $R$

Example schema not in BCNF:

$\text{bor_loan} = (\text{customer_id}, \text{loan_number}, \text{amount})$

because $\text{loan_number} \rightarrow \text{amount}$ holds on $\text{bor_loan}$ but $\text{loan_number}$ is not a superkey.

Decomposing a Schema into BCNF

- Suppose we have a schema $R$ and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.
- We decompose $R$ into:
  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$
- In our example,
  - $\alpha = \text{loan_number}$
  - $\beta = \text{amount}$
  - and $\text{bor_loan}$ is replaced by
    - $(\alpha \cup \beta) = (\text{loan_number}, \text{amount})$
    - $(R - (\beta - \alpha)) = (\text{customer_id}, \text{loan_number})$
BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation.
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that all functional dependencies hold, then that decomposition is dependency preserving.
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as third normal form.

Goals of Normalization

- Goal of normalization: create a set of relational tables that
  - are free of redundant data
  - can be consistently and correctly modified.
- Let $R$ be a relation scheme with a set $F$ of functional dependencies.
- Decide whether a relation scheme $R$ is in "good" form.
- In the case that a relation scheme $R$ is not in "good" form, decompose it into a set of relation scheme $(R_1, R_2, ..., R_n)$ such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving.
How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a database
  
  \[ \text{classes (course, teacher, book)} \]
  
  such that \((c, t, b) \in \text{classes}\) 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).

---

How good is BCNF? (Cont.)

<table>
<thead>
<tr>
<th>course</th>
<th>teacher</th>
<th>book</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>Avi</td>
<td>DB Concepts</td>
</tr>
<tr>
<td>database</td>
<td>Avi</td>
<td>Ullman</td>
</tr>
<tr>
<td>database</td>
<td>Hank</td>
<td>DB Concepts</td>
</tr>
<tr>
<td>database</td>
<td>Hank</td>
<td>Ullman</td>
</tr>
<tr>
<td>database</td>
<td>Sudarshan</td>
<td>DB Concepts</td>
</tr>
<tr>
<td>database</td>
<td>Sudarshan</td>
<td>Ullman</td>
</tr>
<tr>
<td>operating systems</td>
<td>Avi</td>
<td>OS Concepts</td>
</tr>
<tr>
<td>operating systems</td>
<td>Avi</td>
<td>Stallings</td>
</tr>
<tr>
<td>operating systems</td>
<td>Pete</td>
<td>OS Concepts</td>
</tr>
<tr>
<td>operating systems</td>
<td>Pete</td>
<td>Stallings</td>
</tr>
</tbody>
</table>

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if Marilyn is a new teacher that can teach database, two tuples need to be inserted
  
  (database, Marilyn, DB Concepts)
  
  (database, Marilyn, Ullman)
How good is BCNF? (Cont.)

- Therefore, it is better to decompose classes into:

<table>
<thead>
<tr>
<th>course</th>
<th>teacher</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>Avi</td>
</tr>
<tr>
<td>database</td>
<td>Hank</td>
</tr>
<tr>
<td>database</td>
<td>Sudarshan</td>
</tr>
<tr>
<td>operating systems</td>
<td>Avi</td>
</tr>
<tr>
<td>operating systems</td>
<td>Jim</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>teaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>course</td>
</tr>
<tr>
<td>database</td>
</tr>
<tr>
<td>database</td>
</tr>
<tr>
<td>database</td>
</tr>
<tr>
<td>operating systems</td>
</tr>
</tbody>
</table>

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF), which we shall see later.

Functional-Dependency Theory

- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies.
- We then develop algorithms to generate lossless decompositions into BCNF and 3NF.
- We then develop algorithms to test if a decomposition is dependency-preserving.
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 $F^+$.
- 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

- $F = \{ A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H \}$
- 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 $CG \rightarrow I$
  - $CG \rightarrow HI$ by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting 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 shall 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 \( \alpha \), define the closure of \( \alpha \) under \( F \) (denoted by \( \alpha^+ \)) as the set of attributes that are functionally determined by \( \alpha \) under \( F \).

Algorithm to compute \( \alpha^+ \), the closure of \( \alpha \) under \( F \)

\[
\text{result} := \alpha; \\
\text{while (changes to result) do} \\
\quad \text{for each } \beta \rightarrow \gamma \text{ in } F \text{ do} \\
\quad \quad \text{begin} \\
\quad \quad \quad \text{if } \beta \subseteq \text{result then } \text{result} := \text{result} \cup \gamma \\
\quad \quad \text{end}
\]

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. \( \text{result} = AG \)
  2. \( \text{result} = ABCG \) \( (A \rightarrow C \text{ and } A \rightarrow B) \)
  3. \( \text{result} = ABCGH \) \( (CG \rightarrow H \text{ and } CG \subseteq AGBC) \)
  4. \( \text{result} = ABCGHI \) \( (CG \rightarrow I \text{ and } CG \subseteq AGBCH) \)
- Is \( AG \) a candidate key?
  1. Is \( AG \) a super key?
    1. Does \( AG \rightarrow R \) ? \( \equiv \text{Is} (AG)^+ \supseteq R \)
    2. Is any subset of \( AG \) a superkey?
      1. Does \( A \rightarrow R \) ? \( \equiv \text{Is} (A)^+ \supseteq R \)
      2. Does \( G \rightarrow R \) ? \( \equiv \text{Is} (G)^+ \supseteq R \)
Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- **Testing for superkey:**
  - To test if \( \alpha \) is a superkey, we compute \( \alpha^+ \) and check if \( \alpha^+ \) 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 \( \alpha^+ \) by using attribute closure, and then check if it contains \( \beta \).
  - Is a simple and cheap test, and very useful

- **Computing closure of \( F \)**
  - For each \( \gamma \subseteq R \), we find the closure \( \gamma^+ \), 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: \( \{ A \rightarrow B, B \rightarrow C \} \)
  - 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 Attributes

Consider a set $F$ of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in $F$.

- Attribute $A$ is extraneous in $\alpha$ if $A \in \alpha$ and $F$ logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
- Attribute $A$ is extraneous in $\beta$ if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies $F$.

Note: implication in the opposite direction is trivial in each of the cases above, since a “stronger” functional dependency always implies a weaker one.

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 $\alpha$

1. compute $(\{\alpha\} - A)^+ \cup \{\alpha \rightarrow \beta\}$
2. check that $(\{\alpha\} - A)^+$ contains $\beta$; if it does, $A$ is extraneous in $\alpha$

To test if attribute $A \in \beta$ is extraneous in $\beta$

1. compute $\alpha^+$ using only the dependencies in $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$
2. check that $\alpha^+$ contains $A$; if it does, $A$ is extraneous in $\beta$
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 $\beta$
  - 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:
  - $A \rightarrow B$
  - $B \rightarrow C$
Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations $r$ on schema $R$
  
  $$r = \prod_{R_1}(r) \times \prod_{R_2}(r)$$

- A decomposition of $R$ into $R_1$ and $R_2$ is lossless join if and only if at least one of the following dependencies is in $P^+$:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
  - Can be decomposed in two different ways

- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless-join decomposition:
    
    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
  - Dependency preserving

- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Lossless-join decomposition:
    
    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \cap R_2$)
Dependency Preservation

- Let $F_i$ be the set of dependencies $F^*$ that include only attributes in $R_i$.
  - A decomposition is dependency preserving, if
    \[(F_1 \cup F_2 \cup \ldots \cup F_n)^* = F^*\]
  - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Testing for Dependency Preservation

- To check if a dependency $\alpha \to \beta$ is preserved in a decomposition of $R$ into $R_1, R_2, \ldots, R_n$ we apply the following test (with attribute closure done with respect to $F$)
  - \[
  \begin{align*}
  \text{result} &= \alpha \\
  \text{while} \text{ (changes to result) do} \\
  \text{for each} \ R_i \text{ in the decomposition} \\
  t &= (\text{result} \cap R_i)^* \cap R_i \\
  \text{result} &= \text{result} \cup t
  \end{align*}
  \]
  - If \text{result} contains all attributes in $\beta$, then the functional dependency $\alpha \to \beta$ is preserved.
- We apply the test on all dependencies in $F$ to check if a decomposition is dependency preserving.
- This procedure takes polynomial time, instead of the exponential time required to compute $F^*$ and $(F_1 \cup F_2 \cup \ldots \cup F_n)^*$.
Example

- $R = (A, B, C)$
  - $F = \{A \rightarrow B, B \rightarrow C\}$
  - Key = \{A\}
- $R$ is not in BCNF
- Decomposition $R_1 = (A, B), \ R_2 = (B, C)$
  - $R_1$ and $R_2$ in BCNF
  - Lossless-join decomposition
  - Dependency preserving

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
  1. compute $\alpha^+$ (the attribute closure of $\alpha$), and
  2. verify that it includes all attributes of $R$, that is, it is a superkey of $R$.
- **Simplified test:** To check if a relation schema $R$ is in BCNF, it suffices to check only the dependencies in the given set $F$ for violation of BCNF, rather than checking all dependencies in $F^+$.  
  - If none of the dependencies in $F$ causes a violation of BCNF, then none of the dependencies in $F^+$ will cause a violation of BCNF either.
- **However, using only $F$ is incorrect** when testing a relation in a decomposition of $R$
  - Consider $R = (A, B, C, D, E)$, with $F = \{A \rightarrow B, BC \rightarrow D\}$
  - Decompose $R$ into $R_1 = (A, B)$ and $R_2 = (A, C, D, E)$
  - Neither of the dependencies in $F$ contain only attributes from $(A, C, D, E)$ so we might be mislead into thinking $R_2$ satisfies BCNF.
  - In fact, dependency $AC \rightarrow D$ in $F^+$ shows $R_2$ is not in BCNF.
### Testing Decomposition for BCNF

To check if a relation $R_i$ in a decomposition of $R$ is in BCNF,
- Either test $R_i$ for BCNF with respect to the restriction of $F$ to $R_i$ (that is, all FDs in $F^+$ that contain only attributes from $R_i$)
- or use the original set of dependencies $F$ that hold on $R_i$, but with the following test:
  - for every set of attributes $\alpha \subseteq R_i$, check that $\alpha^+$ (the attribute closure of $\alpha$) either includes no attribute of $R_i - \alpha$, or includes all attributes of $R_i$.
  - If the condition is violated by some $\alpha \rightarrow \beta$ in $F$, the dependency $\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$ can be shown to hold on $R_i$, and $R_i$ violates BCNF.
- We use above dependency to decompose $R_i$.

### BCNF Decomposition Algorithm

```plaintext
result := \{ R \};
done := false;
compute $F^+$;
while (not done) do
  if (there is a schema $R_i$ in result that is not in BCNF)
    then begin
      let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds on $R_i$ such that $\alpha \rightarrow R_i$ is not in $F^+$,
      and $\alpha \cap \beta = \emptyset$;
      result := (result $-$ $R_i$) $\cup$ ($R_i$ $-$ $\beta$) $\cup$ ($\alpha$, $\beta$);
    end
  else done := true;
```

Note: each $R_i$ is in BCNF, and decomposition is lossless-join.
Example of BCNF Decomposition

- \( R = (A, B, C) \)
  \( F = \{ A \rightarrow B, B \rightarrow C \} \)
  \( \text{Key} = \{A\} \)
- \( R \) is not in BCNF (\( B \rightarrow C \) but \( B \) is not superkey)
- Decomposition
  - \( R_1 = (B, C) \)
  - \( R_2 = (A, B) \)

Example of BCNF Decomposition

- Original relation \( R \) and functional dependency \( F \)
  \( R = (\text{branch\_name}, \text{branch\_city}, \text{assets}, \text{customer\_name}, \text{loan\_number}, \text{amount}) \)
  \( F = \{ \text{branch\_name} \rightarrow \text{assets} \text{ branch\_city} \}
  \text{loan\_number} \rightarrow \text{amount branch\_name} \}
  \( \text{Key} = \{\text{loan\_number}, \text{customer\_name}\} \)
- Decomposition
  - \( R_1 = (\text{branch\_name}, \text{branch\_city}, \text{assets}) \)
  - \( R_2 = (\text{branch\_name}, \text{customer\_name}, \text{loan\_number}, \text{amount}) \)
  - \( R_3 = (\text{branch\_name}, \text{loan\_number}, \text{amount}) \)
  - \( R_4 = (\text{customer\_name}, \text{loan\_number}) \)
- Final decomposition
  \( R_1, R_3, R_4 \)
BCNF and Dependency Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving

- \( R = (J, K, L) \)
  \( F = \{ JK \rightarrow L, L \rightarrow K \} \)
  Two candidate keys = \( JK \) and \( JL \)
- \( R \) is not in BCNF
- Any decomposition of \( R \) will fail to preserve \( JK \rightarrow L \)
  This implies that testing for \( JK \rightarrow L \) requires a join

Third Normal Form: Motivation

- There are some situations where
  - BCNF is not dependency preserving, and
  - efficient checking for FD violation on updates is important
- Solution: define a weaker normal form, called Third Normal Form (3NF)
  - Allows some redundancy (with resultant problems; we will see examples later)
  - But functional dependencies can be checked on individual relations without computing a join.
  - There is always a lossless-join, dependency-preserving decomposition into 3NF.
3NF Example

- Relation R:
  - $R = (J, K, L)$
  - $F = \{ JK \rightarrow L, L \rightarrow K \}$
  - Two candidate keys: JK and JL
  - $R$ is in 3NF
    - $JK \rightarrow L$  $JK$ is a superkey
    - $L \rightarrow K$  $K$ is contained in a candidate key

Redundancy in 3NF

- There is some redundancy in this schema
- Example of problems due to redundancy in 3NF
  - $R = (J, K, L)$
  - $F = \{ JK \rightarrow L, L \rightarrow K \}$
  - Repetition of information (e.g., the relationship $l_1, k_1$)
  - Need to use null values (e.g., to represent the relationship $l_2, k_2$ where there is no corresponding value for $J$).
Testing for 3NF

- Optimization: Need to check only FDs in \( F \), need not check all FDs in \( F^+ \).
- Use attribute closure to check for each dependency \( \alpha \rightarrow \beta \), if \( \alpha \) is a superkey.
- If \( \alpha \) is not a superkey, we have to verify if each attribute in \( \beta \) is contained in a candidate key of \( R \)
  - this test is rather more expensive, since it involve finding candidate keys
  - testing for 3NF has been shown to be NP-hard
  - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

3NF Decomposition Algorithm

Let \( F_c \) be a canonical cover for \( F \);
\[ i := 0; \]
\[ \text{for each functional dependency } \alpha \rightarrow \beta \text{ in } F_c \text{ do} \]
\[ \text{if none of the schemas } R_j, 1 \leq j \leq i \text{ contains } \alpha \beta \]
\[ \text{then begin} \]
\[ i := i + 1; \]
\[ R_i := \alpha \beta \]
\[ \text{end} \]
\[ \text{if none of the schemas } R_j, 1 \leq j \leq i \text{ contains a candidate key for } R \]
\[ \text{then begin} \]
\[ i := i + 1; \]
\[ R_i := \text{any candidate key for } R; \]
\[ \text{end} \]
\[ \text{return } (R_1, R_2, ..., R_i) \]
3NF Decomposition Algorithm (Cont.)

- Above algorithm ensures:
  - each relation schema $R_i$ is in 3NF
  - decomposition is dependency preserving and lossless-join
  - Proof of correctness is at end of this presentation (click here)

3NF Decomposition: An Example

- Relation schema:
  $\text{cust_banker_branch} = (\text{customer_id, employee_id, branch_name, type})$

- The functional dependencies for this relation schema are:
  1. $\text{customer_id, employee_id} \rightarrow \text{branch_name, type}$
  2. $\text{employee_id} \rightarrow \text{branch_name}$
  3. $\text{customer_id, branch_name} \rightarrow \text{employee_id}$

- We first compute a canonical cover
  - $\text{branch_name}$ is extraneous in the r.h.s. of the 1st dependency
  - No other attribute is extraneous, so we get $F_C =$
    - $\text{customer_id, employee_id} \rightarrow \text{type}$
    - $\text{employee_id} \rightarrow \text{branch_name}$
    - $\text{customer_id, branch_name} \rightarrow \text{employee_id}$
### 3NF Decomposition Example (Cont.)

- The for loop generates following 3NF schema:
  - $(\text{customer_id, employee_id, type})$
  - $(\text{employee_id, branch_name})$
  - $(\text{customer_id, branch_name, employee_id})$
  - Observe that $(\text{customer_id, employee_id, type})$ contains a candidate key of the original schema, so no further relation schema needs be added.
- If the FDs were considered in a different order, with the 2nd one considered after the 3rd,
  - $(\text{employee_id, branch_name})$
  - would not be included in the decomposition because it is a subset of
  - $(\text{customer_id, branch_name, employee_id})$
- Minor extension of the 3NF decomposition algorithm: at end of for loop, detect and delete schemas, such as $(\text{employee_id, branch_name})$, which are subsets of other schemas.
  - result will not depend on the order in which FDs are considered.
- The resultant simplified 3NF schema is:
  - $(\text{customer_id, employee_id, type})$
  - $(\text{customer_id, branch_name, employee_id})$

### Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - the decomposition is lossless
  - the dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - the decomposition is lossless
  - it may not be possible to preserve 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.

Multivalued Dependencies (MVDs)

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

  $t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
  $t_1[\beta] = t_2[\beta]$
  $t_3[R - \beta] = t_1[R - \beta]$
  $t_4[R - \beta] = t_2[R - \beta]$
MVD (Cont.)

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

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>$R - \alpha - \beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>$a_1 \ldots a_i$</td>
<td>$a_{i+1} \ldots a_j$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$a_1 \ldots a_i$</td>
<td>$b_{i+1} \ldots b_j$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$a_1 \ldots a_i$</td>
<td>$a_{i+1} \ldots a_j$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>$a_1 \ldots a_i$</td>
<td>$b_{i+1} \ldots b_j$</td>
</tr>
</tbody>
</table>

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 $Y \rightarrow Z$ ($Y$ multidetermines $Z$) if and only if for all possible relations $r(R)$
  - $< y_1, z_1, w_1 > \in r$ and $< y_1, z_2, w_2 > \in r$
  - then $< y_1, z_1, w_2 > \in r$ and $< y_1, z_2, w_1 > \in r$
- Note that since the behavior of $Z$ and $W$ are identical it follows that $Y \rightarrow Z$ if $Y \rightarrow W$
Example (Cont.)

- In our example:
  \[ \text{course} \rightarrow\rightarrow \text{teacher} \]
  \[ \text{course} \rightarrow\rightarrow \text{book} \]

- The above formal definition is supposed to formalize the notion that given a particular value of \( Y(\text{course}) \) it has associated with it a set of values of \( Z(\text{teacher}) \) and a set of values of \( W(\text{book}) \), 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 $\alpha \rightarrow \rightarrow \beta$
  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 $\alpha \cup \beta = R$)
  - $\alpha$ 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
    \[ \alpha \rightarrow\!\!\!\leftrightarrow (\beta \cap R_i) \]
    where \( \alpha \subseteq R_i \) and \( \alpha \rightarrow\!\!\!\leftrightarrow \beta \) is in \( D^+ \)

4NF Decomposition Algorithm

\[
\text{result}: = \{ R_i \}; \\
\text{done} := \text{false}; \\
\text{compute } D^+; \\
\text{Let } D_i \text{ denote the restriction of } D^+ \text{ to } R_i \\
\text{while (not done)} \\
\quad \text{if (there is a schema } R_i \text{ in result that is not in 4NF)} \text{ then} \\
\quad \quad \text{begin} \\
\quad \quad \quad \text{let } \alpha \rightarrow\!\!\!\leftrightarrow \beta \text{ be a nontrivial multivalued dependency that holds} \\
\quad \quad \quad \text{on } R_i \text{ such that } \alpha \rightarrow R_i \text{ is not in } D_i \text{, and } \alpha \cap \beta \neq \emptyset; \\
\quad \quad \quad \text{result} := (\text{result} - R_i) \cup (R_i - \beta) \cup (\alpha, \beta); \\
\quad \quad \text{end} \\
\quad \text{else done} := \text{true}; \\
\text{Note: each } R_i \text{ is in 4NF, and decomposition is lossless-join}
\]
Example

  
  \[ F = \{ A \rightarrow B \]
  
  \[ B \rightarrow HI \]
  
  \[ CG \rightarrow H \}\]

- **R** is not in 4NF since \( A \rightarrow B \) and \( A \) is not a superkey for \( R \)

- **Decomposition**
  
  a) \( R_1 = (A, B) \) (\( R_1 \) is in 4NF)
  
  b) \( R_2 = (A, C, G, H, I) \) (\( R_2 \) is not in 4NF)
  
  c) \( R_3 = (C, G, H) \) (\( R_3 \) is in 4NF)
  
  d) \( R_4 = (A, C, G, I) \) (\( R_4 \) is not in 4NF)

- Since \( A \rightarrow B \) and \( B \rightarrow HI \), \( A \rightarrow HI \), \( A \rightarrow I \)
  
  e) \( R_5 = (A, I) \) (\( R_5 \) is in 4NF)
  
  f) \( R_6 = (A, C, G) \) (\( R_6 \) is in 4NF)

Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
  
  - lead to project-join normal form (PJNF) (also called fifth normal form)
  
  A class of even more general constraints, leads to a normal form called domain-key normal form.

- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.

- Hence rarely used
Overall Database Design Process

- We have assumed schema $R$ is given
  - $R$ could have been generated when converting E-R diagram to a set of tables.
  - $R$ could have been a single relation containing all attributes that are of interest (called universal relation).
  - Normalization breaks $R$ into smaller relations.
  - $R$ could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
  - Example: an employee entity with attributes `department_number` and `department_address`, and a functional dependency `department_number → department_address`
  - Good design would have made department an entity
- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary
Denormalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying `customer_name` along with `account_number` and `balance` requires join of `account` with `depositor`
- Alternative 1: Use denormalized relation containing attributes of `account` as well as `depositor` with all above attributes
  - faster lookup
  - extra space and extra execution time for updates
  - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined as `account` along `depositor`
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:
  Instead of `earnings` `(company_id, year, amount)`, use
    - Above are in BCNF, but make querying across years difficult and needs new table each year
    - Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
    - Is an example of a crossstab, where values for one attribute become column names
    - Used in spreadsheets, and in data analysis tools
End of Chapter

Based on slides from the text book: Database System Concepts, 5th Ed.