Chapter 8: Relational Database Design

- 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

Combine Schemas?

- Suppose we combine instructor and department into inst_dept
  - (No connection to relationship set inst_dept)
  - Result is possible repetition of information

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
<th>dept_name</th>
<th>building</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>2222</td>
<td>Thomas</td>
<td>70000</td>
<td>History</td>
<td>Painter</td>
<td>120000</td>
</tr>
<tr>
<td>3243</td>
<td>Do Smith</td>
<td>60000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>4566</td>
<td>Katz</td>
<td>70000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>9833</td>
<td>Kim</td>
<td>70000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>6789</td>
<td>Crick</td>
<td>70000</td>
<td>Biology</td>
<td>Watson</td>
<td>90000</td>
</tr>
<tr>
<td>1011</td>
<td>Sinhavasa</td>
<td>60000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>5012</td>
<td>Californy</td>
<td>62000</td>
<td>History</td>
<td>Painter</td>
<td>50000</td>
</tr>
<tr>
<td>8391</td>
<td>Branda</td>
<td>90000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>1515</td>
<td>Mount</td>
<td>60000</td>
<td>Music</td>
<td>Parkard</td>
<td>80000</td>
</tr>
<tr>
<td>3148</td>
<td>Grend</td>
<td>60000</td>
<td>Physics</td>
<td>Watson</td>
<td>70000</td>
</tr>
<tr>
<td>7643</td>
<td>Singh</td>
<td>80000</td>
<td>History</td>
<td>Painter</td>
<td>120000</td>
</tr>
</tbody>
</table>

What is Good Design?

1) Easier: What is Bad Design?

Redundancy is Bad!

- Update Physics Department
  - multiple tuples to update
  - Efficiency + potential for errors
- Delete Physics Department
  - update multiple tuples
  - Efficiency + potential for errors
- Departments without instructor or instructors without departments
  - Need dummy department and dummy instructor
  - Makes aggregation harder and error prone.

A Combined Schema Without Repetition

- Combining is not always bad!
  - Consider combining relations
    - sec_class(sec_id, building, room_number) and
    - section(course_id, sec_id, semester, year)
    - into one relation
    - section(course_id, sec_id, semester, year,
      building, room_number)
    - No repetition in this case
What About Smaller Schemas?

- Suppose we had started with inst_dept. How would we know to split up (decompose) it into instructor and department?
- Write a rule "if there were a schema (dept_name, building, budget), then dept_name would be a candidate key"
- Denote as a functional dependency:
  - dept_name -> building, budget
- In inst_dept, because dept_name is not a candidate key, the building and budget of a department may have to be repeated.
  - This indicates the need to decompose inst_dept
- Not all decompositions are good. Suppose we decompose employee(ID, name, street, city, salary) into employee1 (ID, name) employee2 (name, street, city, salary)
- This indicates the need to decompose inst_dept
- ● This indicates the need to decompose inst_dept
- Not all decompositions are good. Suppose we decompose employee(ID, name, street, city, salary) into employee1 (ID, name) employee2 (name, street, city, salary)
- 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

Example of Lossless-Join Decomposition

- Lossless join decomposition
- Decomposition of R = (A, B, C) R1 = (A, B) R2 = (B, C)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

A B C

\[ r \]

\[ \Pi_{A,B}(r) \cap \Pi_{B,C}(r) \]

\[ \Pi_{A,\sigma(r)} \]

\[ \Pi_{A}(r) \cap \Pi_{B,C}(r) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

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 (R1, R2, ..., Rn) such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on:
  1) Models of dependency between attribute values
     - functional dependencies
     - multivalued dependencies
  2) Concept of lossless decomposition
  3) Normal Forms Based On
     - Atomicity of values
     - Avoidance of redundancy
     - Lossless decomposition

Modeling Dependencies between Attribute Values:

- Functional Dependencies
- Multivalued Dependencies

modified from:
Database System Concepts, 6th Ed.
©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use
Functional Dependencies

- Constraints on the set of legal instances for a relation schema.
- 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.
  - Thus, every key is a functional dependency

Use of Functional Dependencies

- We use functional dependencies to:
  - test relations to see if they are legal under a given set of functional dependencies.
  - 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 instructor may, by chance, satisfy $\text{name} \rightarrow \text{ID}$.

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 $h$ and $b$ of $r$ agree on the attributes $\alpha$, they also agree on the attributes $\beta$. That is,
  - $h[\alpha] = b[\alpha] \Rightarrow h[\beta] = b[\beta]$
- Example: Consider $r(A,B)$ with the following instance of $r$.
  
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</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 $K \rightarrow R$
- $K$ is a candidate key for $R$ if and only if
  - $K \rightarrow R$, and
  - for no $\alpha \subseteq K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:
  inst_dept (ID, name, salary, dept_name, building, budget).
  We expect these functional dependencies to hold:
  - dept_name $\rightarrow$ building
  - ID $\rightarrow$ building
  but would not expect the following to hold:
  - dept_name $\rightarrow$ salary

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 $F^+$.
- $F^+$ is a superset of $F$.

### 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.

- How do we get the initial set of FDs?
  - Semantics of the domain we are modelling
  - Has to be provided by a human (the designer)
- Example:
  - Relation Citizen(SSN, FirstName, LastName, Address)
  - We know that SSN is unique and a person has a unique SSN
  - Thus, SSN $\rightarrow$ FirstName, LastName

### 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 $AG \rightarrow CG$
  - $CG \rightarrow HI$
    - 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 $f$ and $f'$ in $F^+$
    if $f$ and $f'$ 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 more efficient procedure for this task later.
Closure of Functional Dependencies (Cont.)

- Additional rules:
  1. If $\gamma \rightarrow \beta$ holds, then $\alpha \rightarrow \beta$ holds (union)
  2. If $\alpha \rightarrow \beta$ holds, then $\alpha \rightarrow \gamma \beta$ holds (decomposition)
  3. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \gamma \beta$ 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 $F$

  ```
  result := $\alpha$
  while (changes to result) do
    for each $\beta \rightarrow \gamma$ in $\mathcal{F}$ do
      if $\beta \subseteq \alpha$ 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 $A \rightarrow B$)
  3. result := $ABCGH$ ($CG \rightarrow H$ and $CG \subseteq AGBC$)
  4. result := $ABCGHI$ ($CG \rightarrow I$ and $CG \subseteq AGBC$)

- Is $AG$ a candidate key?
  1. Is $AG$ a super key?
    - Does $AG \rightarrow R$? Is $(AG)^+ \supseteq R$
  2. Is any subset of $AG$ a super key?
    - Does $A \rightarrow R$? Is $(A)^+ \supseteq R$
    - 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 $\alpha$ 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 $\delta \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow \delta$.

O(n) Algorithm for Attribute Closure

- Data Structures
  - Enumerate the FDs and attributes
    - int[] c: an integer array with one element per FD that is initialized to the size of the LHS of the FD
  - list<int>[n] rhs: an array of lists with one element per FD. The element stores the numeric ID of the attributes of the FDs RHS
  - list<int>[n] lhs: an array of lists of integers, one element per attribute. The element for each attribute stores the numeric IDs of the FDs that have the attribute in its LHS
  - set<int> aplus: a set storing the attributes currently established to be implied by A
  - stack<int> todo: a stack of attributes to be processed next

- Algorithm
  ```
  initialize c, rhs, lhs, aplus to the emptyset, todo to A
  while (!todo.isEmpty) {
    curA = todo.pop();
    aplus.add(curA); // add curA to result
    for fd in lhs[curA] { // update how many attribute found for LHS
      c[fd]--; // found a LHS attr for fd
      if (c[fd] == 0) {
        remove(lhs[curA], fd); // avoid firing twice
        for new in rhs[fd] { // add implied attributes
          if (aplus[new]) // if attribute is new add to todo
            todo.push(new);
            aplus.add(new);
        }
      }
    }
  }
  ```

O(n) Algorithm for Attribute Closure
Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others.
- For example: \( A \rightarrow BC \) 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.

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 \( \alpha \) is extraneous in \( \alpha \):
  1. Compute \((\alpha - A)^+ \) using the dependencies in \( F \).
  2. Check if \((\alpha - A)^+ \) contains \( \beta \); if it does, \( \alpha \) is extraneous in \( \alpha \).
- To test if attribute \( A \in \beta \) is extraneous in \( \beta \):
  1. Compute \( A^+ \) using only the dependencies in \( F \).
  2. Check if \( A^+ \) contains \( \alpha \); if it does, \( \alpha \) is extraneous in \( \beta \).

Computing a Canonical Cover

- \( R = \{ A, B, C \} \)
  - \( F = \{ A \rightarrow BC, B \rightarrow C, AB \rightarrow C \} \)
  - Combine \( A \rightarrow BC \) and \( A \rightarrow B \) into \( A \rightarrow BC \).
  - \( A \) is extraneous in \( AB \rightarrow C \).
  - Check if the result of deleting \( A \) from \( AB \rightarrow C \) is implied by the other dependencies.
  - Yes: \( B \rightarrow C \) is already present!
  - Set is now \( \{ A \rightarrow BC, B \rightarrow C \} \).
- \( C \) is extraneous in \( A \rightarrow BC \).
  - Check if \( A \rightarrow BC \) 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: \( A \rightarrow B, B \rightarrow C \).

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

Canonical Cover

- A canonical cover for \( F \) is a set of dependencies \( F' \) such that:
  - \( F \) logically implies all dependencies in \( F' \), and
  - \( F \) logically implies all dependencies in \( F' \), and
  - No functional dependency in \( F' \) contains an extraneous attribute, and
  - Each left side of functional dependency in \( F' \) is unique.
- To compute a canonical cover for \( F \):
  1. Use the union rule to replace any dependencies in \( F \) that can be inferred even after deleting \( C \).
  2. Repeat until an extraneous attribute is found, delete it from \( F \).
  3. Use the union rule to replace any dependencies in \( F \).
Testing for Dependency Preservation

- To check if a dependency \( \alpha \rightarrow \beta \) 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 \( F \))
  - result = \( \alpha \)
  - for each \( R_i \) in the decomposition
    - \( t = (\text{result} \cap R_i) \cap R \)
    - result = result \( \cup t \)
  - If result contains all attributes in \( \beta \), then the functional dependency \( \alpha \rightarrow \beta \) is preserved.
- We apply the test on all dependencies in \( F \) to check if a decomposition is dependency preserving
- This procedure (attribute closure) takes polynomial time, instead of the exponential time required to compute \( F^+ \) and \( (F_1 \cup F_2 \cup \ldots \cup F_n)^+ \)

Example

- Let \( F \) 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.

Dependency Preservation

- For the case of \( R = (R_1, R_2) \), we require that for all possible relation instances \( r \) on schema \( R \)
  - \( r = \Pi_{\alpha \in F} (r) \) \( \Pi_{\alpha \notin F} (r) \)
- A decomposition of \( R \) into \( R_1 \) and \( R_2 \) is lossless join if at least one of the following dependencies is in \( F^+: \)
  - \( R_1 \cap R_2 \rightarrow R_1 \)
  - \( R_1 \cap R_2 \rightarrow R_2 \)
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies.
Normal Forms

Goals of Normalization

- 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 schemes \( R_1, R_2, \ldots, R_n \) such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving.

First Normal Form

- A 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 (revisited in Chapter 22 of the textbook: Object Based Databases).

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.

Second Normal Form

- A relation schema R in 1NF is in second normal form (2NF) if
  - No non-prime attribute depends on parts of a candidate key.
  - An attribute is non-prime if it does not belong to any candidate key for R.

So Far

- Theory of dependencies
- Decompositions and ways to check whether they are “good”
  - Lossless
  - Dependency preserving
- What is missing?
  - Define what constitutes a good relation
    - Normal forms
  - How to check for a good relation
    - Test normal forms
  - How to achieve a good relation
    - Translate into normal form
    - Involves decomposition
Second Normal Form Example

- R(A,B,C,D)
  - A,B → C,D
  - A → C
  - B → D

- (A,B) is the only candidate key
- R is not in 2NF, because A→C where A is part of a candidate key and C is not part of a candidate key

- Interpretation: R(A,B,C,D) is Advisor(InstrSSN, StudentCWID, InstName, StudentName)
  - Indication that we are putting stuff together that does not belong together

Second Normal Form Interpretation

- Why is a dependency on parts of a candidate key bad?
  - That is why is a relation that is not in 2NF bad?
  - 1) A dependency on part of a candidate key indicates potential for redundancy

- Example: Advisor(InstrSSN, StudentCWID, InstName, StudentName)
  - StudentCWID ® StudentName

- If a student is advised by multiple instructors we record his name several times

- 2) A dependency on parts of a candidate key shows that some attributes are unrelated to other parts of a candidate key
  - That means the table should be split

2NF is What We Want?

- Instructor(Name, Salary, DepName, DepBudget) = I(A,B,C,D)
  - A → B,C,D
  - C → D

- {Name} is the only candidate key
- I is in 2NF

- However, as we have seen before I still has update redundancy that can cause update anomalies
  - We repeat the budget of a department if there is more than one instructor working for that department

Third Normal Form

- A relation schema R is in third normal form (3NF) if for all:
  - \( a \rightarrow b \) in \( F \)
    - at least one of the following holds:
      - \( a \rightarrow b \) is trivial (i.e., \( b \subseteq a \))
      - \( a \) is a superkey for \( R \)
      - Each attribute \( A \) in \( b - a \) is contained in a candidate key for \( R \).

- Alternatively:
  - Every attribute depends directly on a candidate key, i.e., for every attribute \( A \) there is a dependency \( X \rightarrow A \), but no dependency \( Y \rightarrow A \) where \( Y \) is not a candidate key

3NF Example

- Instructor(Name, Salary, DepName, DepBudget) = I(A,B,C,D)
  - A → B,C,D
  - C → D

- {Name} is the only candidate key
- I is in 2NF

- I is not in 3NF

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 \( a \rightarrow b \), if \( a \) is a superkey.

- If \( a \) is not a superkey, we have to verify if each attribute in \( b \) 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
The functional dependencies for this relation schema are:

- If none of the schemas $R_i$, $1 \leq i \leq j$, contains $\alpha \beta$, then $R_j \rightarrow \alpha \beta$
- If none of the schemas $R_i$, $1 \leq j \leq i$, contains a candidate key for $R_j$, then $R_j \rightarrow \alpha$

3NF Decomposition Algorithm

Let $F_c$ be a canonical cover for $F$.

for each functional dependency $\alpha \rightarrow \beta$ in $F_c$ do

if none of the schemas $R_i$, $1 \leq j \leq i$, contains $\alpha \beta$ then begin

$\alpha \beta \rightarrow \alpha$

end

if none of the schemas $R_i$, $1 \leq j \leq i$, contains a candidate key for $R_j$ then begin

$R_j \rightarrow \alpha$

end

end

return $(R_1, R_2, \ldots, R_n)$

3NF Decomposition: An Example

- Relation schema:
  - $cust_banker_branch = (customer_id, employee_id, branch_name, type)$

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

- We first compute a canonical cover
  - $branch_name$ is extraneous in the r.h.s. of the 1st dependency
  - No other attribute is extraneous, so we get $F_c =$
    - $customer_id, employee_id \rightarrow type$
    - $employee_id \rightarrow branch_name$
    - $customer_id, branch_name \rightarrow employee_id$

3NF Decomposition Example (Cont.)

- The for loop generates following 3NF schema:
  - $(customer_id, employee_id, type)$
  - $(employee_id, branch_name)$
  - $(customer_id, branch_name, employee_id)$

- Observe that $(customer_id, employee_id, type)$ contains a candidate key of the original schema, so no further relation schema needs to be added.
- At end of for loop, detect and delete schemas, such as $(employee_id, branch_name)$, which are subsets of other schemas
- Result will not depend on the order in which FDs are considered
- The resultant simplified 3NF schema is:
  - $(customer_id, employee_id, type)$
  - $(customer_id, branch_name, employee_id)$

Another 3NF Example

- Relation $dept_advisor$:
  - $dept_advisor = (s_id, i_id, dept_name)$
  - $F = (s_id, dept_name \rightarrow i_id, i_id \rightarrow dept_name)$
- Two candidate keys: $s_id, dept_name, and i_id, s_id$
- $R$ is in 3NF
  - $s_id, dept_name \rightarrow i_id, s_id$
  - $dept_name$ is a superkey
  - $i_id \rightarrow dept_name$
- $dept_name$ is contained in a candidate key

Redundancy in 3NF

- There is some redundancy in this schema $dept_advisor(s_id, i_id, dept_name)$
- Example of problems due to redundancy in 3NF
  - $R = (i_id, K, L)$
  - $F = (A \rightarrow L, K \rightarrow K')$

- Repetition of information (e.g., the relationship $i, K$)
  - $(i_id, dept_name)$
- Need to use null values (e.g., to represent the relationship $i, K$ where there is no corresponding value for $j$)
  - $(i_id, dept_name)$ if there is no separate relation mapping instructors to departments
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 $R$

Example schema not in BCNF:

```
instr_dept (ID, name, salary, dept_name, building, budget)
```

because $\text{dept_name} \rightarrow \text{building, budget}$ holds on $\text{instr_dept}$ but $\text{dept_name}$ is not a superkey

BCNF and Dependency Preservation

- If a relation is in BCNF it is in 3NF
- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- Because it is not always possible to achieve both BCNF and dependency preservation, we usually consider normally third normal form.

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 $\alpha$).
  2. Verify that it includes all attributes of $R$ that is, $\alpha$ 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, simplified test 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 $R_i = (A, B)$ 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 misled into thinking $R$ satisfies BCNF.
  - In fact, dependency $AC \rightarrow D$ in $F$ shows $R$ is not in BCNF.

Testing Decomposition for BCNF

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

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:
    - $(\alpha \cup \beta)$
    - $(R \setminus (\beta \setminus \alpha))$
  - In our example,
    - $\alpha = \text{dept_name}$
    - $\beta = \text{building, budget}$
    - and $\text{instr_dept}$ is replaced by
      - $(\alpha \cup \beta) = (\text{dept_name}, \text{building, budget})$
      - $(R \setminus (\beta \setminus \alpha)) = (\text{ID, name, salary, dept_name})$

BCNF Decomposition Algorithm

- Suppose for some $\alpha \rightarrow \beta$ a nontrivial dependency that holds in $R$ such that $\alpha \rightarrow \beta$ is not in $F^*$
  - and $\alpha \setminus \beta = \emptyset$.
- Let $\alpha = \text{dept_name}$ and $\beta = \text{building, budget}$.
- Compute $\alpha^+$ and $\beta^+$.
- If there is a relation $R$ in $F$ such that $\alpha \rightarrow \beta$ is not in $F^*$
  - then begin
    - Let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds in $R$ such that $\alpha \rightarrow \beta$ is not in $F^*$
    - and $\alpha \setminus \beta = \emptyset$.
    - Compute $\text{result} := (R \setminus (\beta \setminus \alpha))$.
  - end
  - else begin
    - Compute $\alpha^+$ and $\beta^+$.
    - Compute $\text{result} := (R \setminus (\beta \setminus \alpha))$.
  - end

Note: each $R$ is in BCNF, and decomposition is lossless-join.
Consider a relation sufficiently normalized. There are database schemas in BCNF that do not seem to be in BCNF. How do we know this? Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples (99999, David, 981-992-3443) and (99999, William, 981-992-3443).

Example of BCNF Decomposition

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

Example of BCNF Decomposition (Cont.)

- $R = \{JK, K, L\}$
- $F = \{JK \rightarrow L, L \rightarrow K\}$
- 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.

Example of BCNF Decomposition

- $R = \{\text{course_id, title, dept_name, credits, sec_id, semester, year, building, room_number, capacity, time_slot_id}\}$
- Functional dependencies:
  - course_id $\rightarrow$ title, dept_name, credits
  - building, room_number $\rightarrow$ capacity
  - course_id, sec_id, semester, year $\rightarrow$ building, room_number, time_slot_id
- A candidate key (course_id, sec_id, semester, year).
- BCNF Decomposition:
  - course_id $\rightarrow$ title, dept_name, credits holds
  - but course_id is not a superkey.
  - We replace class $\rightarrow$:
    - course(course_id, title, dept_name, credits)
    - class-1 (course_id, sec_id, semester, year, building, room_number, capacity, time_slot_id)

How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized.
- Consider a relation inst_info (ID, child_name, phone)
  - where an instructor may have more than one phone and can have multiple children

<table>
<thead>
<tr>
<th>ID</th>
<th>child_name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>99999</td>
<td>David</td>
<td>512-555-1234</td>
</tr>
<tr>
<td>99999</td>
<td>David</td>
<td>512-555-4321</td>
</tr>
<tr>
<td>99999</td>
<td>William</td>
<td>512-555-1234</td>
</tr>
<tr>
<td>99999</td>
<td>William</td>
<td>512-555-4321</td>
</tr>
</tbody>
</table>

How good is BCNF? (Cont.)

- There are no non-trivial functional dependencies and therefore the relation is in BCNF.
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples (99999, David, 981-992-3443) and (99999, William, 981-992-3443).
How good is BCNF? (Cont.)

Therefore, it is better to decompose `inst_info` into:

<table>
<thead>
<tr>
<th>ID</th>
<th>child_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>99999</td>
<td>David</td>
</tr>
<tr>
<td>99999</td>
<td>David</td>
</tr>
<tr>
<td>99999</td>
<td>William</td>
</tr>
<tr>
<td>99999</td>
<td>William</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>99999</td>
<td>512-555-1234</td>
</tr>
<tr>
<td>99999</td>
<td>512-555-4321</td>
</tr>
<tr>
<td>99999</td>
<td>512-555-1234</td>
</tr>
<tr>
<td>99999</td>
<td>512-555-4321</td>
</tr>
</tbody>
</table>

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

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.

Summary Normal Forms

- BCNF -> 3NF -> 2NF -> 1NF
- 1NF
  - atomic attributes
- 2NF
  - no non-trivial dependencies of non-prime attributes on parts of the key
- 3NF
  - no transitive non-trivial dependencies on the key
- BCNF
  - only non-trivial dependencies on a superkey

Design Goals Revisited

- 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, (and currently not supported by any of the widely used databases!)
- 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

- Suppose we record names of children, and phone numbers for instructors:
  - `inst_child(ID, child_name)`
  - `inst_phone(ID, phone_number)`
- If we were to combine these schemas to get
  - `inst_info(ID, child_name, phone_number)`
  - Example data:
    - (99999, David, 512-555-1234)
    - (99999, David, 512-555-4321)
    - (99999, William, 512-555-1234)
    - (99999, William, 512-555-4321)
  - This relation is in BCNF
    - Why?
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 of tuples \( t_i \) and \( t_j \) in \( r \) such that \( t_i[\alpha] = t_j[\alpha] \), there exist tuples \( t_i \) and \( t_k \) in \( r \) such that:

\[
\begin{align*}
    t_i[\alpha] &= t_j[\alpha] = t_k[\alpha] \\
    t_i[\beta] &= t_j[\beta] = t_k[\beta] \\
    t_i[R - \beta] &= t_j[R - \beta] = t_k[R - \beta]
\end{align*}
\]

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 \( r \).

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 \( r(R) \)

\[
\begin{align*}
    &< y, z, w > \in r \text{ and } < y, z, w > \in r \\
    \text{then} &< y, z, w > \in r
\end{align*}
\]

Note that since the behavior of \( Z \) and \( W \) are identical it follows that

\( Y \rightarrow Z \text{ if } Y \rightarrow W \)

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 \( \alpha \rightarrow \beta \)

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).

Example (Cont.)

In our example:

\[
\begin{align*}
    &ID \rightarrow \text{child_name} \\
    &ID \rightarrow \text{phone_number}
\end{align*}
\]

The above formal definition is supposed to formalize the notion that given a particular value of \( Y \) it has associated with it a set of values of \( Z \) (\( \text{child_name} \)) and a set of values of \( W \) (\( \text{phone_number} \)), and these two sets are in some sense independent of each other.

Note:

- If \( Y \rightarrow Z \) then \( Y \rightarrow Z \)

- Indeed we have \( (\text{in above notation}) Z_1 = Z_2 \)

The claim follows.

Example (Cont.)

In our example:

\[
\begin{align*}
    &ID \rightarrow \text{child_name} \\
    &ID \rightarrow \text{phone_number}
\end{align*}
\]

The above formal definition is supposed to formalize the notion that given a particular value of \( Y \) it has associated with it a set of values of \( Z \) (\( \text{child_name} \)) and a set of values of \( W \) (\( \text{phone_number} \)), and these two sets are in some sense independent of each other.

Note:

- If \( Y \rightarrow Z \) then \( Y \rightarrow Z \)

- Indeed we have \( (\text{in above notation}) Z_1 = Z_2 \)

The claim follows.
**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 $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ 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 $R$ is the set $D'$ consisting of
  - All functional dependencies in $D$ that include only attributes of $R$
  - All multivalued dependencies of the form $\alpha \rightarrow \beta \cap R$ where $\alpha \subseteq R$ and $\alpha \rightarrow \beta$ is in $D^*$

**4NF Decomposition Algorithm**

```plaintext
result = (R);
done = false;
compute D;
Let D denote the restriction of D^* to R;
while (not done) begin
  if (there is a schema R_i in result that is not in 4NF) then
    let α → β be a nontrivial multivalued dependency that holds on R such that α → R_i is not in D_i and α ∪ β = β_i;
    result = (result ∪ (R_i ∪ (α → β_i)));
    end
  else done = true;
end
Note: each R_i is in 4NF, and decomposition is lossless-join
```

**Example**

- $F = \{ A \rightarrow B, B \rightarrow H, CG \rightarrow H \}$
- $R$ is not in 4NF since $A \rightarrow B$ and $A$ is not a superkey for $R$
- Decomposition
  - a) $R_1 = (A, B)$
  - b) $R_2 = (A, C, G, H, I)$
  - c) $R_3 = (C, G, H)$
  - d) $R_4 = (A, C, G, H)$
  - e) $R_5 = (A, I)$
  - f) $R_6 = (A, C, G)$

**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

**Final Thoughts on Design Process**

modified from:
Database System Concepts, 6th Ed.
©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use
Overall Database Design Process

- We have assumed schema \( R \) is given
  - \( R \) could have been generated when converting an ER 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 ER diagram is carefully designed, identifying all entities correctly, the tables generated from the ER 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, name and building, and a functional dependency department \( \rightarrow \) building
  - 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 prereqs along with course, id, and title requires join of course with prereq
- Alternative 1: Use denormalized relation containing attributes of course as well as prereq 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 course prereq
  - 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:
  - 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 crosstab, where values for one attribute become column names
  - Used in spreadsheets, and in data analysis tools

Recap

- Functional and Multi-valued Dependencies
  - Axioms
  - Closure
  - Minimal Cover
  - Attribute Closure
  - Redundancy and lossless decomposition
- Normal Forms
  - 1NF, 2NF, 3NF
  - BCNF
  - 4NF, 5NF

End of Chapter

modified from: Database System Concepts, 6th Ed.
©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use
Proof of Correctness of 3NF Decomposition Algorithm

Claim: if a relation $R$ is in the decomposition generated by the above algorithm, then $R$ satisfies 3NF.

- Let $R$ be generated from the dependency $a \rightarrow b$.
- Let $\gamma \rightarrow b$ be any non-trivial functional dependency on $R_i$. (We need only consider FDs whose right-hand side is a single attribute.)
- Now, $b$ can be in either $\gamma$ or $a$ but not in both. Consider each case separately.

Case 1: $B$ in $\gamma$.
- Since $a$ is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.
- In fact, we cannot show that $\gamma$ is a superkey.
- This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.

Case 2: $B$ is in $a$.
- Since $a$ is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.
- In fact, we cannot show that $\gamma$ is a superkey.
- This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.

Figure 8.02

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
<th>dept_name</th>
<th>building</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>2222</td>
<td>Einstein</td>
<td>95000</td>
<td>Physics</td>
<td>Watson</td>
<td>70000</td>
</tr>
<tr>
<td>1212</td>
<td>Wu</td>
<td>90000</td>
<td>Finance</td>
<td>Painter</td>
<td>120000</td>
</tr>
<tr>
<td>3234</td>
<td>El Said</td>
<td>60000</td>
<td>History</td>
<td>Taylor</td>
<td>50000</td>
</tr>
<tr>
<td>4565</td>
<td>Katz</td>
<td>75000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>10000</td>
</tr>
<tr>
<td>9834</td>
<td>Kim</td>
<td>80000</td>
<td>Elec. Eng.</td>
<td>Taylor</td>
<td>80000</td>
</tr>
<tr>
<td>7656</td>
<td>Crick</td>
<td>72000</td>
<td>Biology</td>
<td>Watson</td>
<td>90000</td>
</tr>
<tr>
<td>1010</td>
<td>Srinivasan</td>
<td>65000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>10000</td>
</tr>
<tr>
<td>3583</td>
<td>Califere</td>
<td>62000</td>
<td>History</td>
<td>Painter</td>
<td>50000</td>
</tr>
<tr>
<td>8343</td>
<td>Brandt</td>
<td>92000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>10000</td>
</tr>
<tr>
<td>1353</td>
<td>Mozart</td>
<td>40000</td>
<td>Music</td>
<td>Packard</td>
<td>80000</td>
</tr>
<tr>
<td>3346</td>
<td>Gold</td>
<td>87000</td>
<td>Physics</td>
<td>Watson</td>
<td>70000</td>
</tr>
<tr>
<td>7654</td>
<td>Singh</td>
<td>80000</td>
<td>Finance</td>
<td>Painter</td>
<td>120000</td>
</tr>
</tbody>
</table>
### Figure 8.17

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1</td>
<td>b_1</td>
<td>c_1</td>
</tr>
<tr>
<td>a_2</td>
<td>b_2</td>
<td>c_2</td>
</tr>
<tr>
<td>a_3</td>
<td>b_3</td>
<td>c_3</td>
</tr>
</tbody>
</table>