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

What is Good Design?

1) Easier: What is Bad Design?

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>Address</th>
<th>Budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>2222</td>
<td>Evans</td>
<td>90000</td>
<td>Physics</td>
<td>Watson</td>
<td>70000</td>
</tr>
<tr>
<td>1221</td>
<td>Wu</td>
<td>90000</td>
<td>Finance</td>
<td>Painter</td>
<td>120000</td>
</tr>
<tr>
<td>3234</td>
<td>Elbad</td>
<td>80000</td>
<td>History</td>
<td>Painter</td>
<td>50000</td>
</tr>
<tr>
<td>4510</td>
<td>Katz</td>
<td>70000</td>
<td>Comp Sci</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>9654</td>
<td>Kim</td>
<td>50000</td>
<td>Elec Eng</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>5765</td>
<td>Chick</td>
<td>72000</td>
<td>Biology</td>
<td>Watson</td>
<td>50000</td>
</tr>
<tr>
<td>0110</td>
<td>Stivers (math)</td>
<td>60000</td>
<td>Comp Sci</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>8888</td>
<td>Calfert</td>
<td>62000</td>
<td>History</td>
<td>Painter</td>
<td>50000</td>
</tr>
<tr>
<td>3832</td>
<td>Braun</td>
<td>92000</td>
<td>Comp Sci</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>5333</td>
<td>Morton</td>
<td>60000</td>
<td>Music</td>
<td>Parkard</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>2843</td>
<td>Seng</td>
<td>80000</td>
<td>Finance</td>
<td>Painter</td>
<td>120000</td>
</tr>
</tbody>
</table>

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

A Lossless Decomposition

- Lossless-join decomposition means splitting a table in a way so that we do not lose information.
- That means we should be able to reconstruct the original table from the decomposed table using joins.

Example of Lossless-Join Decomposition

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

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.
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.

Functional Dependencies (Cont.)

- Let $R$ be a relation schema $\alpha \subseteq R$ and $\beta \subseteq R$.
- The functional dependency $\alpha \rightarrow \beta$ holds on $R$ if and only if for any legal relations $r(R)$, whenever any two tuples $t_1$ and $t_2$, if $r$ 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 $r$.

  - $A \rightarrow B$ holds on $r$.

  - On this instance, $A \rightarrow B$ does hold, but $B \rightarrow A$ does not 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
  - $K \rightarrow \beta$ for no $\beta \subset K$.
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

  $$\text{inst_dept: (ID, name, salary, dept_name, building, budget)}.$$

  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

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 $F$.
    - Specifying 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 $name \rightarrow ID$.

Functional Dependencies (Cont.)

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

- $A \rightarrow B$ does NOT hold, but $B \rightarrow A$ does hold.
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 \( F \).

Prove Additional Implications

- Prove or disprove the following rules from Armstrong’s axioms:
  1) \( A \rightarrow B \) and \( C \) implies \( A \rightarrow B \) and \( A \rightarrow C \)
  2) \( A \rightarrow B \) and \( A \rightarrow C \) implies \( A \rightarrow B \) and \( A \rightarrow C \)
  3) \( A, B \rightarrow C \) implies \( A \rightarrow C \)
  4) \( A \rightarrow B \) and \( C \rightarrow D \) implies \( A, C \rightarrow B, D \)

Function-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

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 more efficient procedure for this task later.
Closure of Functional Dependencies (Cont.)

- Additional rules:
  - If \( \alpha \rightarrow \beta \gamma \) 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 \( \gamma \rightarrow \lambda \) holds, then \( \alpha \gamma \rightarrow \lambda \) holds (pseudotransitivity)

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

Example of Attribute Set Closure

- \( R = (A, B, C, G, H, I) \)
- \( F = \{(A \rightarrow B), (A = C), (CG \rightarrow H), (CG \rightarrow I), (B \rightarrow H)\} \)
- \( (AG)^+ \)

1. result = AG
2. result = ABCG \( (A \rightarrow C \text{ and } A \rightarrow B) \)
3. result = ABCGH \( (CG \rightarrow H \text{ and } CG \subset AGBC) \)
4. result = ABCGHI \( (CG \rightarrow I \text{ and } CG \subset AGBCH) \)

Is \( AG \) a candidate key?
  - Does \( AG \rightarrow R? \Rightarrow \text{Is } (AG)^+ \subset R \)
  - Is \( AG \) a super key?
    - Does \( AG \rightarrow R? \Rightarrow \text{Is } (AG)^+ \subset R \)
  - Is any subset of \( AG \) a super key?
    - Does \( A \rightarrow RM \Rightarrow \text{Is } (AM)^+ \subset R \)
    - Does \( G \rightarrow RM \Rightarrow \text{Is } (GM)^+ \subset R \)

O(n) Algorithm for Attribute Closure

- Data Structures
  - Enumerate the FDs and attributes
    - int[] e: an integer array with one element per FD that is initialized to the size of the LHS of the FD
    - list<int>[] 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>[] 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

O(1) Algorithm for Attribute Closure

- Algorithm

  ```java
  int curA = todo.pop(); // 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 newA in rhs[fd] { // add implied attributes
        if (aplus[newA]) // if attribute is new add to todo
          todo.push(newA);
        aplus.add(newA); // add implied attributes
      }
    }
  }
  ```

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 \( F \)

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

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 \( \beta \subset \alpha^+ \).
  - That is, we compute \( \alpha^+ \) by using attribute closure, and then check if it contains \( \beta \).
- It is a simple and cheap test, and very useful
- Computing closure of \( F \)
  - For each \( \gamma \subset R \), we find the closure \( \gamma^+ \), and for each \( S \subset \gamma^+ \), we output a functional dependency \( \gamma \rightarrow S \).
Canonical Cover

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

Extraneous Attributes

- Consider a set \( F \) of functional dependencies and the functional dependency \( \alpha \rightarrow \beta \) in \( F \).
  - Attribute \( A \) is extraneous in \( \alpha \rightarrow \beta \) if \( A \) is extraneous in \( \alpha \) and \( F \) logically implies \( (F - (\alpha \rightarrow \beta)) \cup (\alpha \rightarrow \beta) \).
  - 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 \( A \) is extraneous in \( \alpha \):
  1. Compute \( (\alpha - A)^+ \) using the dependencies in \( F \).
  2. Check that \( (\alpha - A)^+ \) contains \( \beta \); if it does, \( A \) is extraneous in \( \alpha \).
- To test if \( \alpha \rightarrow \beta \) 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 \( \alpha \); if it does, \( A \) is extraneous in \( \beta \).

Canonical Cover

- A canonical cover for \( F \) is a set of dependencies \( F_c \) such that:
  - \( F_c \) logically implies all dependencies in \( F \).
  - \( F_c \) logically implies all dependencies in \( F \).
  - 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 \( F \):
  1. Use the union rule to replace any dependencies in \( F \) with \( (\alpha \rightarrow \beta) \) and \( (\alpha \rightarrow \beta) \) such that \( A \rightarrow C \) with \( \alpha \rightarrow \beta - B \) and \( \beta \rightarrow A \). \( \alpha \rightarrow \beta \).
  2. Find a functional dependency \( \alpha \rightarrow \beta \) with all extraneous attributes and if it exists, delete it from \( \alpha \rightarrow \beta \).
  3. Repeat 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.

Lossless Join-Decomposition

- Dependency Preservation

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

- **Theory of dependencies**
  - What is missing?
    - When is a decomposition lossless
      - Lossless-join decomposition
        - Dependencies on the input are preserved
  - What else 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

Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relation instances $r$ on schema $R$
  - 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.

Example

- $R = (A, B, C)$
  - $F = (A \rightarrow B, B \rightarrow C)$
    - Can be decomposed in two different ways
      - $R_1 = (A, B), R_2 = (B, C)$
        - Lossless-join decomposition:
          - $R_1 \cap R_2 = (B)$ and $B \rightarrow BC$
          - Dependency preserving
      - $R_1 = (A, B), R_2 = (A, C)$
        - Lossless-join decomposition:
          - $R_1 \cap R_2 = (A)$ and $A \rightarrow AB$
          - Not dependency preserving
            (cannot check $B \rightarrow C$ without computing $R_1$ and $R_2$)

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 = $\alpha$
    - while (changes to result do)
      - for each $R_i$ in the decomposition
        - $result = result \cup \{ R_i \}$
        - result is a subset of the dependencies
      - 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 $(F_1 \cup F_2 \cup \ldots \cup F_n)^*$.

Dependency Preservation

- Let $F_1$ be the set of dependencies $F$ that include only attributes in $R_1$.
  - 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.

Example

- $R = (A, B, C)$
  - $F = (A \rightarrow B, B \rightarrow C)$
    - Key = $\{A\}$
    - Decomposition $R_1 = (A, B), R_2 = (B, C)$
      - Lossless-join decomposition
      - Dependency preserving
Normal Forms

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

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 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
Second Normal Form Example

- R(A,B,C,D)
  - A,B → C
  - 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, InstrName, 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
    - Advisor(InstrSSN, StudentCWID, InstrName, 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
  - 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:
  - α → β in P
  - at least one of the following holds:
    - α → β is trivial (i.e., β ∈ α)
    - α is a superkey for R
    - Each attribute A in β → α is contained in a candidate key for R.
      (NOTE: each attribute may be in a different candidate key)
  - Alternatively,
    - Every attribute depends directly on a candidate key, i.e., for every attribute A there is a dependency X → A, but no dependency Y → A
      where Y is not a candidate key

3NF Example

- Instructor(Name, Salary, DepName, DepBudget) = I(A,B,C,D)
  - A → B,C
  - 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 P:
  - Use attribute closure to check for each dependency α → β in P.
    - If α is not a superkey, we have to verify if each attribute in β 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 \( F \).

for each functional dependency \( \alpha \rightarrow \beta \) in \( F_c \), do

if none of the schemas \( R_j \) s.t. \( \alpha \subseteq s \) contains \( \alpha \beta \)

then begin

\( j \leftarrow j + 1; \)

\( R_j \rightarrow \alpha \beta; \)

end

if none of the schemas \( R_j \) s.t. \( \alpha \subseteq s \) contains a candidate key for \( R \)

then begin

\( j \leftarrow j + 1; \)

\( R_j \rightarrow \) any candidate key for \( R; \)

end

/* Optionally, remove redundant relations */

repeat

if any schema \( R \) is contained in another schema \( R_j \)

then begin

\( F \leftarrow delete R \); \)

\( j \leftarrow j + 1; \)

return \( (R_1, R_2, ..., R_j) \)

3NF Decomposition Example (Cont.)

The for loop generates following 3NF schema:

\[
\begin{align*}
& (\text{customer_id, employee_id, type}) \\
& (\text{customer_id, branch_name, employee_id, id})
\end{align*}
\]

Observe that \( (\text{customer_id, employee_id, type}) \) contains a candidate key of the original schema, so no further relation schema needs be added

At end of for loop, detect and delete schemas, such as \( (\text{employee_id}) \), which are subsets of other schemas

result will not depend on the order in which FDs are considered

The resultant simplified 3NF schema is:

\[
\begin{align*}
& (\text{customer_id, employee_id, type}) \\
& (\text{customer_id, branch_name, employee_id})
\end{align*}
\]

Another 3NF Example

Relation dept_advisor:

\[
\begin{align*}
& (\text{s_ID, i_ID, dept_name}) \\
& F = (\text{s_ID, dept_name} \rightarrow \text{i_ID, dept_name})
\end{align*}
\]

Two candidate keys: \( s_ID, \text{dept_name}, \) and \( i_ID, s_ID \)

\( R \) is in 3NF

Example of problems due to redundancy in 3NF

\[
\begin{align*}
& R = (\text{i_ID, K, L}) \\
& F = (\text{J} \rightarrow \text{K})
\end{align*}
\]

repetition of information (e.g., the relationship \( i_1, k_1 \))

\( (i_ID, \text{dept_name}) \) if there is no separate relation mapping instructors to departments

Redundancy in 3NF

There is some redundancy in this schema dept_advisor \( (\text{s_ID, i_ID, dept_name}) \)

Example of problems due to redundancy in 3NF

\[
\begin{align*}
& R = (\text{i_ID, K, L}) \\
& F = (\text{J} \rightarrow \text{K})
\end{align*}
\]

repeat of information (e.g., the relationship \( i_1, k_1 \))

\( (i_ID, \text{dept_name}) \) 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:

- $\text{instr_dept} (\text{id}, \text{name}, \text{salary}, \text{dept_name}, \text{building}, \text{budget})$
- because $\text{dept_name} \rightarrow \text{building}, \text{budget}$
- 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 $\alpha \cup \beta$ (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.
- 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 $R_1 = (A, B)$ 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 misled into thinking $R_1$ satisfies BCNF.
- In fact, dependency $AC \rightarrow D$ in $P$ shows $R_1$ 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 $\alpha \cup \beta$ (the attribute closure of $\alpha$) either includes no attribute of $R_i \cup \alpha$, or includes all attributes of $R_i$.
    - If the condition is violated by some $\alpha \rightarrow \beta$ in $F$, the dependency $\alpha \rightarrow (\alpha \cup \beta)$ holds on $R_i$ and $R_i$ violates BCNF.
    - We use above dependency to decompose $R_i$.

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

BCNF Decomposition Algorithm

```python
result := (R);
done := false;
compute F*;
while (not done) do
  if (there is a schema R 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 $\alpha \rightarrow \beta$ is not in $F*$.
      and $\alpha \cap (\alpha \cup \beta) = \emptyset$;
      result := (result $\cup (R_i - \beta \rightarrow \alpha) \cup (\alpha \cup \beta))$;
    end;
  else done := true;
end;
```

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

- $R = (A, B, C)$
- $F = (A \rightarrow B)$
- 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

- $R = (Course_id, Title, Dept_name, Credits, Sec_id, Semester, Year, Building, Room_number, Capacity, Time_slot_id)$
- Functional dependencies:
  - $Course_id$ $\rightarrow$ Title, Dept_name, Credits
  - Building, Room_number $\rightarrow$ Capacity
  - Sec_id, Course_id $\rightarrow$ Semester, Year 
  - Building, Room_number, Time_slot_id
- $A$ candidate key $(Course_id, Sec_id, Semester, Year)$
- BCNF Decomposition:
  - $R_1 = (Course_id, Title, Dept_name, Credits)
  - $R_2 = (Building, Room_number, Capacity, Time_slot_id)$
- $R$ is not in BCNF
- A candidate key $(Course_id, Sec_id, Semester, Year)$ is not a superkey for $R$
- We replace $R_1$ by:
  - $L = JK, J, L$
  - $R = (JK, JK, L)$
  - $F = (JK \rightarrow L)$
- Two candidate keys are 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

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 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) 99999
  - (99999, William, 981-992-3443) 99999
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>

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

<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>

How good is BCNF? (Cont.)

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.

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 and 4NF, 5NF

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
  
  \( \alpha \rightarrow\rightarrow \beta \)

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

  \[
  \begin{align*}
  t_1[\alpha] &= t_2[\alpha] \\
  t_1[\beta] &= t_2[\beta] \\
  t_1[\beta] &= t_2[\beta] \\
  t_1[\beta] &= t_2[\beta] \\
  t_1[\beta] &= t_2[\beta]
  \end{align*}
  \]

- We say that \( \alpha \rightarrow\rightarrow \beta \) holds on \( R \).

**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 \( Y \rightarrow\rightarrow Z \) (\( Y \) multidetermines \( Z \))

  if and only if for all possible relations \( r(R) \)

  \[
  \begin{align*}
  &<y_1, z_1, w_1 > \in r \land <y_2, z_2, w_2 > \in r \\
  \text{then} &<y_1, z_1, w_1 > \in r \land <y_2, z_2, w_2 > \in r
  \end{align*}
  \]

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

**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 \( r \).

**MVD (Cont.)**

- Tabular representation of \( \alpha \rightarrow\rightarrow \beta \)

<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_1 \ldots a_i )</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>( a_1 \ldots a_i )</td>
<td>( a_1 \ldots a_i )</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>( a_1 \ldots a_i )</td>
<td>( a_1 \ldots a_i )</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>( a_1 \ldots a_i )</td>
<td>( a_1 \ldots a_i )</td>
</tr>
</tbody>
</table>

**Example (Cont.)**

- In our example:

  \( \text{ID} \rightarrow\rightarrow \text{child\_name} \)

  \( \text{ID} \rightarrow\rightarrow \text{phone\_number} \)

  The above formal definition is supposed to formalize the notion that given a particular value of \( Y \) (ID) it has associated with it a set of values of \( Z \) (child\_name) and a set of values of \( W \) (phone\_number), 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.

**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 \( \alpha \rightarrow \beta \), 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 \( \alpha \cap \beta = \emptyset \))
  - \( \alpha \) 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
    \[ \alpha \rightarrow \beta \cap R_i \]
    where \( \alpha \subseteq R \) and \( \alpha \rightarrow \beta \) is in \( D' \)

4NF Decomposition Algorithm

- result = \( \{R\} \);
- done = false;
- compute \( D^* \);
- Let \( D_i \) denote the restriction of \( D^* \) to \( R_i \)
- while (not done)
  - if (there is a schema \( R \) in result that is not in 4NF) then
    - let \( \alpha \rightarrow \beta \) be a nontrivial multivalued dependency that holds
    - result := (result - \( R \)) \cup (\( R \times \beta \) \cup (\( \alpha \times \beta \));
  - else done := true;
- Note: each \( R \) is in 4NF, and decomposition is lossless-join

Further Normal Forms

- Join dependencies generalize multivalued dependencies
  - lead to project-join normal form (P-JNF) (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

Example

- \( R \rightarrow (A, B, C, G, H, I) \)
- \( F(A \rightarrow B) \)
- \( A \rightarrow H \)
- \( C \rightarrow G \)

- \( R \) 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, decompose into \( R_3 \) and \( R_4 \))
  - c) \( R_3 = (C, G, H) \) (\( R_3 \) is in 4NF)
  - d) \( R_4 = (A, C, G, I) \) (\( R_4 \) is not in 4NF, decompose into \( R_5 \) and \( R_6 \))
  - \( A \rightarrow \beta \) and \( B \rightarrow \gamma \) \( \Rightarrow A \rightarrow H \) (MVD transitivity), and
    - and hence \( A \rightarrow I \) (MVD restriction to \( R_6 \))
  - e) \( R_5 = (A, I) \) (\( R_5 \) is in 4NF)
  - f) \( R_6 = (A, C, G) \) (\( R_6 \) is in 4NF)

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

©Silberschatz, Korth and Sudarshan
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_name → building.
- Good design would have made department an entity.

Denormalization for Performance

- May want to use non-normalized schema for performance.
- For example, displaying prereq 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
Proof of Correctness of 3NF Decomposition Algorithm

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

Correctness of 3NF Decomposition Algorithm

- 3NF decomposition algorithm is dependency preserving (since there is a relation for every FD in $F_c$
- Decomposition is lossless
  - A candidate key ($C$) is in one of the relations $R_i$ in decomposition
  - Closure of candidate key under $F_i$ must contain all attributes in $R_i$
- Follow the steps of attribute closure algorithm to show there is only one tuple in the join result for each tuple in $R_i$

Correctness of 3NF Decomposition Algorithm (Cont’d.)

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

- Let $R_i$ be generated from the dependency $\alpha \rightarrow \beta$.
- 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 $\beta$ or $\alpha$ but not in both. Consider each case separately.

Case 1: If $B$ in $\beta$:

- If $\gamma$ is a superkey, the 2nd condition of 3NF is satisfied.
- Otherwise $\gamma$ must contain some attribute not in $\gamma$.
  - Since $\gamma \rightarrow B$ is in $F^+$ it must be derivable from $F_c$ by using attribute closure on $\gamma$.
  - Attribute closure not have used $\alpha \rightarrow \beta$. If it had been used, $\gamma$ must be contained in the attribute closure of $\gamma$, which is not possible, since we assumed $\gamma$ is not a superkey.
  - Now, using $\alpha \rightarrow (\beta \setminus B)$ and $\gamma \rightarrow B$ (since $\gamma \subseteq \alpha$, and $B \in \gamma$ since $\gamma \rightarrow B$ is non-trivial) we can derive $\alpha \rightarrow B$.
  - Then, $B$ is extraneous in the right-hand side of $\alpha \rightarrow \beta$; which is not possible since $\alpha \rightarrow \beta$ is in $F_c$.
  - Thus, if $B$ is in $\beta$ then $\gamma$ must be a superkey, and the second condition of 3NF must be satisfied.

Q.E.D.

Correctness of 3NF Decomposition Algorithm (Cont’d.)

- Case 2: $B$ is in $\alpha$.
  - Since $\alpha$ 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>12121</td>
<td>Wu</td>
<td>90000</td>
<td>Finance</td>
<td>Painter</td>
<td>120000</td>
</tr>
<tr>
<td>32543</td>
<td>El Said</td>
<td>60000</td>
<td>History</td>
<td>Painter</td>
<td>50000</td>
</tr>
<tr>
<td>43365</td>
<td>Katz</td>
<td>75000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>80000</td>
<td>Elec. Eng.</td>
<td>Taylor</td>
<td>85000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>72000</td>
<td>Bioeng</td>
<td>Watson</td>
<td>90000</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>60000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>36563</td>
<td>Calitri</td>
<td>62000</td>
<td>History</td>
<td>Painter</td>
<td>50000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>92000</td>
<td>Comp. Sci.</td>
<td>Taylor</td>
<td>100000</td>
</tr>
<tr>
<td>13131</td>
<td>Mozart</td>
<td>40000</td>
<td>Music</td>
<td>Packard</td>
<td>80000</td>
</tr>
<tr>
<td>33436</td>
<td>Gold</td>
<td>87000</td>
<td>Physics</td>
<td>Watson</td>
<td>70000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>80000</td>
<td>Finance</td>
<td>Painter</td>
<td>120000</td>
</tr>
</tbody>
</table>
Figure 8.03

Figure 8.04

Figure 8.05

Figure 8.06

Figure 8.14

Figure 8.15
**Figure 8.17**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
</tr>
<tr>
<td>a4</td>
<td>b4</td>
<td>c4</td>
</tr>
</tbody>
</table>