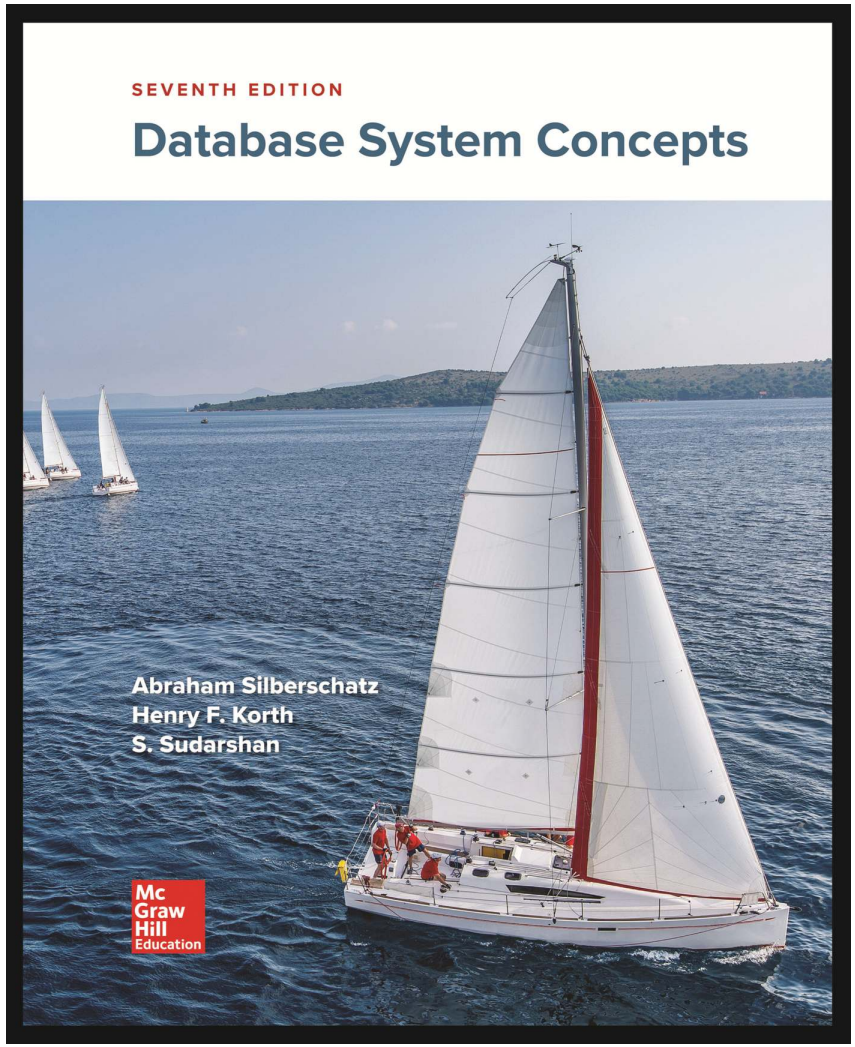
Database System Concepts, 7^{th} Edition
Chapter 7: Relational Database Design

Silberschatz, Korth and Sudarshan

May 4, 2025

Database System Concepts



Content has been extracted from *Database System Concepts*, Seventh Edition, by Silberschatz, Korth and Sudarshan. Mc Graw Hill Education. 2019.
Visit `https://db-book.com/`.

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

Features of Good Relational Designs

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *dept_name*, *credits*)
instructor(*ID*, *name*, *dept_name*, *salary*)
section(*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
student(*ID*, *name*, *dept_name*, *tot_cred*)
takes(*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
advisor(*s_ID*, *i_ID*)
time_slot(*time_slot_id*, *day*, *start_time*, *end_time*)
prereq(*course_id*, *prereq_id*)

Figure 7.1 Database schema for the university example.

Features of Good Relational Designs

- Suppose we combine *instructor* and *department* into *in_dep*, which represents the natural join on the relations *instructor* and *department*.

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Figure 7.2 The *in_dep* relation.

- There is repetition of information
- Need to use `null` values (if we add a new department with no instructors)

Decomposition

- The only way to avoid the repetition-of-information problem in the *in_dep* schema is to decompose it into two schemas –*instructor* and *department* schemas.
- Not all decompositions are good. Suppose we decompose:

  `employee(ID, name, street, city, salary)`

  into

  `employee1 (ID, name)`

  `employee2 (name, street, city, salary)`

  The problem arises when we have two employees with the same name.
- 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

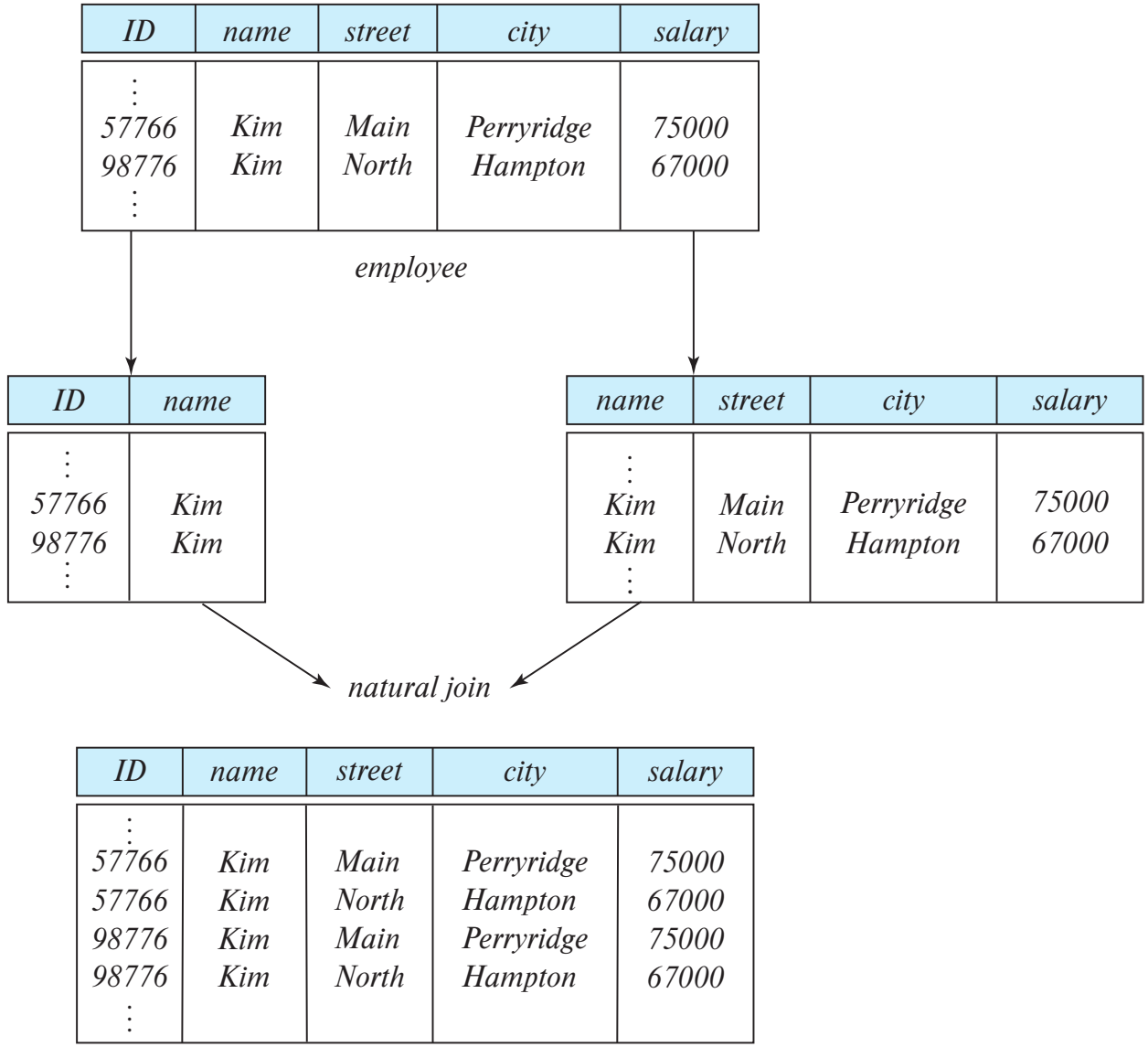


Figure 7.3 Loss of information via a bad decomposition.

Lossless Decomposition

- Let R be a relation schema and let R_1 and R_2 form a decomposition of R. That is $R = R_1 \bigcup R_2$.
- We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing R with the two relation schemas $R_1 \bigcup R_2$.
- Formally,

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

- And, conversely a decomposition is lossy if

$$r \subset \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$$

Example of Lossless Decomposition

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

$$R_1 = (A, B); R_2 = (B, C)$$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

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

A	B	C
α	1	A
β	2	B

Normalization Theory

- 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 set of relations $\{R_1, R_2, \cdots, R_n\}$ such that,
  - Each relation is in good form.
  - The decomposition is a lossless decomposition.
- Our theory is based on:
  1. functional dependencies.
  2. multivalued dependencies.

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

Functional Dependencies

- There are usually a variety of constraints (rules) on the data in the real world.
- For example, some of the constraints that are expected to hold in a university database are:
  - Students and instructors are uniquely identified by their ID.
  - Each student and instructor has only one name.
  - Each instructor and student is (primarily) associated with only one department.
  - Each department has only one value for its budget, and only one associated building.

Functional Dependencies (Cont.)

- An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation.
- A legal instance of a database is one where all the relation instances are legal instances.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a key.

Functional Dependencies Definition

- Let R be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

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

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

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

A	B
1	4
1	5
3	7

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

Closure of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F.
  - 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^+.

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

Keys and Functional Dependencies

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

  `in_dep(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:
  - 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.
  - To 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

$$name \rightarrow ID.$$

Trivial Functional Dependencies

- 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 $\beta \subseteq \alpha$.

Lossless Decomposition

- We can use functional dependencies to show when certain decomposition are lossless.
- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless decomposition if at least one of the following dependencies is in F^+:

$$R_1 \bigcap R_2 \rightarrow R_1$$

$$R_1 \bigcap R_2 \rightarrow R_2$$

- 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[1]

- $R = (A, B, C)$

$$F = \{A \rightarrow B$$
$$B \rightarrow C\}$$

- $R_1 = (A, B)$; $R_2 = (B, C)$
  Lossless decomposition:
  $R_1 \bigcap R_2 = \{B\}$ and $B \rightarrow BC$ holds.
- $R_1 = (A, B)$; $R_2 = (A, C)$
  Lossless decomposition:
  $R_1 \bigcap R_2 = \{A\}$ and $A \rightarrow AC$ holds.

[1] Note: $B \rightarrow BC$ is a shorthand notation for $B \rightarrow \{B, C\}$.

Dependency Preservation

- Testing functional dependency constraints each time the database is updated can be costly,
- It is useful to design the database in a way that constraints can be tested efficiently.
- If testing a functional dependency can be done by considering just one relation, then the cost of testing this constraint is low.
- When decomposing a relation it is possible that it is no longer possible to do the testing without having to perform a Cartesian Produced.
- A decomposition that makes it computationally hard to enforce functional dependency is said to be NOT **dependency preserving**.

Dependency Preservation Example

- Consider a schema:

  `dept_advisor(s_ID, i_ID, department_name)`

- With function dependencies:

$$i\_ID \rightarrow dept\_name$$
$$s\_ID, dept\_name \rightarrow i\_ID$$

- In the above design we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship.
- To fix this, we need to decompose *dept_advisor*.
- Any decomposition will not include all the attributes in:

$$s\_ID, dept\_name \rightarrow i\_ID$$

- Thus, the composition NOT be dependency preserving.

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

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., $\beta \subseteq \alpha$)
  - α is a superkey for R.

Boyce-Codd Normal Form (Cont.)

- Example schema that is not in BCNF:

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

  because :
  - $dept\_name \rightarrow building, budget$
    - holds on `in_dep`
    - but...
  - $dept\_name$ is not a superkey.
- When decompose `in_dept` into `instructor` and `department`:
  - `instructor` is in BCNF.
  - `department` is in BCNF.

Decomposing a Schema into BCNF

- Let R be a schema that is not in BCNF. Let $\alpha \rightarrow \beta$ be the Fuctional Dependency that causes a violation of BCNF.
- We decompose R into:
  - $(\alpha \bigcup \beta)$
  - $(R - (\beta - \alpha))$
- In our example of `in_dep`,
  - $\alpha = dept\_name$
  - $\beta = building, budget$

  and `in_dep` is replaced by
  - $(\alpha \bigcup \beta) = (dept\_name, building, budget)$
  - $(R - (\beta - \alpha)) = (ID, name, salary, dept\_name)$

BCNF and Dependency Preservation

- It is not always possible to achieve both BCNF and dependency preservation.
- Consider a schema:

  `dept_advisor(s_ID, i_ID, department_name)`
- With function dependencies:

$$\begin{aligned} i\_ID &\rightarrow dept\_name \\ s\_ID, dept\_name &\rightarrow i\_ID \end{aligned}$$

- `dept_advisor` is not in BCNF:
  - i_ID is not a superkey.
- Any decomposition of `dept_advisor` will not include all the attributes in:

$$s\_ID, dept\_name \rightarrow i\_ID$$

- Thus, the composition is NOT be dependency preserving.

Third Normal Form

- A relation schema R is in **third normal form (3NF)** if for all:

  $$\alpha \rightarrow \beta \text{ in } F^+$$

  at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$).
  - α is a superkey for R.
  - Each attribute A in $\beta - \alpha$ is contained in a candidate key[2] for R.
- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

[2]NOTE: each attribute may be in a different candidate key.

3NF Example

- Consider a schema:

  `dept_advisor(s_ID, i_ID, dept_name)`

- With function dependencies:

$$i\_ID \rightarrow dept\_name$$
$$s\_ID, dept\_name \rightarrow i\_ID$$

- Two candidate keys = $\{s\_ID, dept\_name\}$, $\{s\_ID, i\_ID\}$
- We have seen before that `dept_advisor` is not in BCNF.
- R, however, is in 3NF:
  - s_ID, $dept\_name$ is a superkey,
  - $i\_ID \rightarrow dept\_name$ and i_ID is NOT a superkey, but:
    - $dept\_name - i\_ID = dept\_name$ and
    - $dept\_name$ is contained in a candidate key.

Redundancy in 3NF

- Consider the schema R below, which is in 3NF:
  - $R = (J, K, L)$,
  - $F = \{JK \rightarrow L, L \rightarrow K\}$,
  - And an instance table:

J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
`null`	l_2	k_2

- What is wrong with the table?

Redundancy in 3NF

- Consider the schema R below, which is in 3NF:
  - $R = (J, K, L)$,
  - $F = \{JK \rightarrow L, L \rightarrow K\}$,
  - And an instance table:

J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
`null`	l_2	k_2

- What is wrong with the table?
  - Repetition of information,
  - Need to use `null` values (e.g., to represent the relationship l_2, k_2 where there is no corresponding value for J)

Comparison of BCNF and 3NF

- Advantages to 3NF over BCNF. It is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation.
- Disadvantages to 3NF.
  - We may have to use `null` values to represent some of the possible meaningful relationships among data items.
  - There is the problem of repetition of information.

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 scheme $\{R_1, R_2, \cdots, R_n\}$ such that:
  - Each relation scheme is in good form,
  - The decomposition is a lossless decomposition,
  - Preferably, the decomposition should be dependency preserving.

How good is BCNF?

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

  `inst_info(ID, child_name, phone)`

  where an instructor may have more than one phone and can have multiple children.
- Instance of `inst_info`

ID	child_name	phone
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	William	512-555-4321

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

Higher Normal Forms

- It is better to decompose `inst_info` into:
  - `inst_child`

ID	child_name
99999	David
99999	William

  - `inst_phone`

ID	phone
99999	512-555-1234
99999	512-555-4321

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

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

Functional-Dependency Theory Roadmap

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

Closure of a Set of Functional Dependencies

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

Closure of a Set of Functional Dependencies

- We can compute F^+, the closure of F, by repeatedly applying **Armstrong's Axioms**:
  - **Reflexive rule**: if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$,
  - **Augmentation rule**: if $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$,
  - **Transitivity rule**: if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$.
- These rules are:
  - **sound** – generate only functional dependencies that actually hold, and
  - **complete** – generate all functional dependencies that hold.

Example of F^+

- $R = (A, B, C, G, H, I)$

$$\begin{aligned} F = \{A &\rightarrow B \\ A &\rightarrow C \\ CG &\rightarrow H \\ CG &\rightarrow I \\ B &\rightarrow H\} \end{aligned}$$

- Some members of F^+
  - $A \rightarrow H$ by transitivity from $A \rightarrow B$ and $B \rightarrow H$.
  - $AG \rightarrow I$ by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$.
  - $CG \rightarrow HI$ by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity.

Closure of a Set of Functional Dependencies (Cont.)

- Additional rules:
  - **Union rule**: If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
  - **Decomposition rule**: If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
  - **Pseudotransitivity rule**: If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.
- The above rules can be inferred from Armstrong's axioms.

Procedure for Computing F^+

- To compute[3] the closure of a set of functional dependencies F:

$F^+ = F$
apply the reflexivity rule /* Generates all trivial dependencies */
repeat
    for each functional dependency f in F^+
        apply the augmentation rule 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
            add the resulting functional dependency to F^+
until F^+ does not change any further

Figure 7.7 A procedure to compute F^+.

[3]NOTE: We shall see an alternative procedure for this task later.

Closure of Attribute Sets

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

---

result := α;
repeat
  for each functional dependency β → γ **in** F **do**
    begin
      if β ⊆ *result* **then** *result* := *result* ∪ γ;
    end
until (*result* does not change)

Figure 7.8 An algorithm to compute α^+, the closure of α under F.

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$

$$\begin{aligned} F = \{A &\rightarrow B \\ A &\rightarrow C \\ CG &\rightarrow H \\ CG &\rightarrow I \\ B &\rightarrow H\} \end{aligned}$$

- $(AG)^+$
  1. result $= AG$.
  2. result $= ABCG (A \rightarrow C$ and $A \rightarrow B)$.
  3. result $= ABCGH (CG \rightarrow H$ and $CG \subseteq AGBC)$.
  4. result $= ABCGHI (CG \rightarrow I$ and $CG \subseteq AGBCH)$.
- Is AG a candidate key?
  1. Is AG a super key?
     1.1 Does $AG \rightarrow R$? == Is $R \supseteq (AG)^+$.
  2. Is any subset of AG a superkey?
     2.1 Does $A \rightarrow R$? == Is $R \supseteq (A)^+$.
     2.2 Does $G \rightarrow R$? == Is $R \supseteq (G)^+$.
     2.3 In general: check for each subset of size $n - 1$.

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

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

Canonical Cover

- When a user updates a relation, the database system must ensure that all functional dependencies in the set F are still satisfied.
- If any FD is violated, the update is rolled back.
- To minimize the effort of checking for violations, the system can test a simplified set of F with the same closure as the given set.
- This simplified set is termed the **canonical cover**.
- To define canonical cover we must first define **extraneous attributes**:
  - An attribute of a functional dependency in F is extraneous if we can remove it without changing F^+.

Extraneous Attributes

- Removing an attribute from the left side of a functional dependency could make it a stronger constraint.
  - For example, if we have $AB \rightarrow C$ and remove B, we get the possibly stronger result $A \rightarrow C$. It may be stronger because $A \rightarrow C$ logically implies $AB \rightarrow C$, but $AB \rightarrow C$ does not, on its own, logically imply $A \rightarrow C$.
- But, depending on what our set F of functional dependencies happens to be, we may be able to remove B from $AB \rightarrow C$ safely.
  - For example, suppose that: $F = \{AB \rightarrow C, A \rightarrow D, D \rightarrow C\}$
  - Then we can show that F logically implies $A \rightarrow C$, making extraneous in $AB \rightarrow C$.

Extraneous Attributes (Cont.)

- Removing an attribute from the right side of a functional dependency could make it a weaker constraint.
  - For example, if we have $AB \rightarrow CD$ and remove C, we get the possibly weaker result $AB \rightarrow D$. It may be weaker because using just $AB \rightarrow D$, we can no longer infer $AB \rightarrow C$.
- But, depending on what our set F of functional dependencies happens to be, we may be able to remove C from $AB \rightarrow CD$ safely.
  - For example, suppose that: $F = \{AB \rightarrow CD, A \rightarrow C\}$.
  - Then we can show that even after replacing $AB \rightarrow CD$ by $AB \rightarrow D$, we can still infer $AB \rightarrow C$ and thus $AB \rightarrow CD$.

Extraneous Attributes

- An attribute of a functional dependency in F is extraneous if we can remove it without changing F^+.
- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.
  - **Remove from the left side**: Attribute A is extraneous in α if:
    - $A \in \alpha$ and
    - F logically implies $(F - \{\alpha \rightarrow \beta\}) \bigcup \{(\alpha - A) \rightarrow \beta\}$.
  - **Remove from the right side**: Attribute A is extraneous in β if:
    - $A \in \beta$ and
    - The set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \bigcup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.

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

Examples of Extraneous Attributes

- Let
$$F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$$
- To check if C is extraneous in $AB \rightarrow CD$, we:
  - Compute the attribute closure of AB under $F' = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$,
    - Start with the initial set: $AB^+ = \{A, B\}$
    - Apply the functional dependencies:
      $AB \rightarrow D$: Since $A, B \in AB^+$, add D: $AB^+ = \{A, B, D\}$
      $A \rightarrow E$: Since $A \in AB^+$, add E: $AB^+ = \{A, B, D, E\}$
      $E \rightarrow C$: Since $E \in AB^+$, add C: $AB^+ = \{A, B, D, E, C\}$

Examples of Extraneous Attributes

- Let
$$F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$$
- To check if C is extraneous in $AB \rightarrow CD$, we:
  - Compute the attribute closure of AB under $F' = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$,
  - The closure is $ABCDE$, which includes CD,
  - This implies that C is extraneous.

Canonical Cover

- A canonical cover for F is a set of dependencies F_c such that:
  - F logically implies all dependencies in F_c, and
  - F_c logically implies all dependencies in F, and
  - No functional dependency in F_c contains an extraneous attribute, and
  - Each left side of functional dependency in F_c is unique. That is, there are no two dependencies in F_c
    - $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ such that
    - $\alpha_1 = \alpha_2$

Canonical Cover

To compute a canonical cover for F:

$F_c = F$
repeat
  Use the union rule to replace any dependencies in F_c of the form
    $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$.
  Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous
    attribute either in α or in β.
    /* Note: the test for extraneous attributes is done using F_c, not F */
  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ in F_c.
until (F_c does not change)

Figure 7.9 Computing canonical cover.

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

Example: Computing a Canonical Cover

- $R = (A, B, C)$

$$F = \{A \to BC$$
$$B \to C$$
$$A \to B$$
$$AB \to C\}$$

- Combine $A \to BC$ and $A \to B$ into $A \to BC$
  - Set is now $\{A \to BC, B \to C, AB \to C\}$
- A is extraneous in $AB \to C$
  - Check if the result of deleting A from $AB \to C$ is implied by the other dependencies ($B \to C$ is already present!).
  - Set is now $\{A \to BC, B \to C\}$
- C is extraneous in $A \to BC$
  - Check if $A \to C$ is logically implied by $A \to B$ and the other dependencies (using transitivity on $A \to B$ and $B \to C$).
- The canonical cover is:

$$A \to B$$
$$B \to C$$

Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i.
  - A decomposition is dependency preserving, if

$$(F_1 \bigcup F_2 \bigcup \cdots \bigcup F_n)^+ = F^+$$

- Using the above definition, testing for **dependency preservation** take exponential time.
- Not that if a decomposition is NOT dependency preserving then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Dependency Preservation (Cont.)

- Let F be the set of dependencies on schema R and let $R_1, R_2, \ldots, R_n$ be a decomposition of R.
- The restriction of F to R_i is the set F_i of all functional dependencies in F^+ that include only attributes of R_i.
- Since all functional dependencies in a restriction involve attributes of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation.
- Note that the definition of restriction uses all dependencies in F^+, not just those in F.
- The set of restrictions $F_1, F_2, \ldots, F_n$ is the set of functional dependencies that can be checked efficiently.

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 = α
repeat
    for each R_i in the decomposition
        t = (result ∩ R_i)^+ ∩ R_i
        result = result ∪ t
until (result does not change)
```

- We apply the test on all dependencies in F to check if a decomposition is dependency preserving.
- This procedure takes polynomial time, instead of the exponential time required to compute F^+ and $(F_1 \bigcup F_2 \bigcup \ldots F_n)+$.

Example

- $R = (A, B, C)$

$$F = \{A \rightarrow B$$
$$B \rightarrow C\}$$
$$Key = \{A\}$$

- R is not in BCNF.
- Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$:
  - R_1 and R_2 in BCNF.
  - Lossless-join decomposition.
  - Dependency preserving.

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF:
  1. compute α^+ (the attribute closure of α), and
  2. verify that it includes all attributes of R, that is, it is a superkey of R.
- **Simplified test**: To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than checking all dependencies in F^+.
  - If none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either ...

Testing for BCNF (Cont.)

- 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 mislead into thinking R_2 satisfies BCNF.
  - In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.

Testing Decomposition for BCNF

- To check if a relation Ri 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, but with the following test:
    - for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.
  - If the condition is violated by some $\alpha \rightarrow \beta$ in F^+, the dependency:

$$\alpha \rightarrow (\alpha^+ - \alpha) \bigcap R_i$$

    can be shown to hold on R_i, and R_i violates BCNF.
  - We use above dependency to decompose R_i.

BCNF Decomposition Algorithm

```
result := {R};
done := false;
while (not done) do
   if (there is a schema R_i in result that is not in BCNF)
     then begin
             let α → β be a nontrivial functional dependency that holds
             on R_i such that α+ does not contain R_i and α ∩ β = ∅;
             result := (result − R_i) ∪ (R_i − β) ∪ ( α, β);
          end
     else done := true;
```

Figure 7.11 BCNF decomposition algorithm.

Note: each R_i is in BCNF, and decomposition is lossless-join.

Example of BCNF Decomposition

- `class (course_id, title, dept_name, credits, sec_id, semester, year, building, room_number, capacity, time_slot_id)`

$$F = \{course\_id \rightarrow title, dept\_name, credits$$
$$building, room\_number \rightarrow capacity$$
$$course\_id, sec\_id, semester, year \rightarrow building, room\_number, time\_slot\_id\}$$

- 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` by:
    - `course(course_id, title, dept_name, credits)`
    - `class-1 (course_id, sec_id, semester, year, building, room_number, capacity, time_slot_id)`

Example of BCNF Decomposition (Cont.)

- course is in BCNF.
  - How do we know this?
- $building, room\_number \rightarrow capacity$ holds on `class-1`
  - but $\{building, room\_number\}$ is not a superkey for `class-1`.
  - We replace `class-1` by:
    - `classroom(building, room_number, capacity)`
    - `section (course_id, sec_id, semester, year, building, room_number, time_slot_id)`

Third Normal Form

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

3NF Example

- Relation `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

Testing for 3NF

- Need to check only FDs in F, need not check all FDs in F^+.
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- 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;
$i := 0$;
for each functional dependency $\alpha \rightarrow \beta$ in F_c
    $i := i + 1$;
    $R_i := \alpha\,\beta$;
if none of the schemas $R_j, j = 1, 2, \ldots, i$ contains a candidate key for R
  then
    $i := i + 1$;
    $R_i :=$ any candidate key for R;
/* Optionally, remove redundant relations */
repeat
    if any schema R_j is contained in another schema R_k
      then
        /* Delete R_j */
        $R_j := R_i$;
        $i := i$ - 1;
until no more R_js can be deleted
return $(R_1, R_2, \ldots, R_i)$

Figure 7.12 Dependency-preserving, lossless decomposition into 3NF.

3NF Decomposition Algorithm (Cont.)

- Above algorithm ensures:
  - each relation schema R_i is in 3NF
  - decomposition is dependency preserving and lossless-join
  - Proof of correctness is in the Book.

3NF Decomposition: An Example

- Relation schema:

  ```
  cust_banker_branch = (customer_id, employee_id, branch_name,
      type)
  ```

- The functional dependencies for this relation schema are:

$$customer\_id, employee\_id \rightarrow branch\_name, type$$
$$employee\_id \rightarrow branch\_name$$
$$customer\_id, branch\_name \rightarrow employee\_id$$

- We first compute a canonical cover:
  - *branch_name* is extraneous in the r.h.s. of the 1^{st} 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: An Example

- 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 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)

Comparison of BCNF and 3NF

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

Design Goals

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

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

Multivalued Dependencies (MVDs)

- 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

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

$$\alpha \twoheadrightarrow \beta$$

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

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

MVD – Tabular representation

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

Figure 7.13 Tabular representation of $\alpha \twoheadrightarrow \beta$.

MVD (Cont.)

- Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets: Y, Z, W.
- We say that $Y \twoheadrightarrow Z$ (Y multidetermines Z) if and only if for all possible relations $r(R)$:

$$< y_1, z_1, w_1 > \in r \text{ and } < y_1, z_2, w_2 > \in r$$

then

$$< y_1, z_1, w_2 > \in r \text{ and } < y_1, z_2, w_1 > \in r$$

- Note that since the behavior of Z and W are identical it follows that

$$Y \twoheadrightarrow Z \text{ if } Y \twoheadrightarrow W$$

Example

- In our example:
  $ID \twoheadrightarrow child\_name$
  $ID \twoheadrightarrow phone\_number$
- 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 $Y \twoheadrightarrow Z$
  - Indeed we have (in above notation) $Z_1 = Z_2$
    The claim follows.

Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:
  1. To test relations to **determine** whether they are legal under a given set of functional and multivalued dependencies.
  2. To specify **constraints** on the set of legal relations. We shall 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:

  $$\text{If } \alpha \rightarrow \beta, \text{ then } \alpha \twoheadrightarrow \beta$$

  That is, every functional dependency is also a multivalued dependency.
- The closure D^+ of D is the set of all functional and multivalued dependencies logically implied by D.
  - We can compute $D+$ from D, using the formal definitions of functional dependencies and multivalued dependencies.
  - We can manage with such reasoning for very simple multivalued dependencies, which seem to be most common in practice
  - For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules (Section 28.1.1).

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

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 $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \twoheadrightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \bigcup \beta = R$).
  - α is a superkey for schema R.
- If a relation is in 4NF it is in BCNF.

Restriction of Multivalued Dependencies

The restriction of D to R_i is the set D_i consisting of:

- All functional dependencies in D^+ that include only attributes of R_i
- All multivalued dependencies of the form:

$$\alpha \twoheadrightarrow (\beta \bigcap R_i)$$

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

4NF Decomposition Algorithm

result := {*R*};
done := false;
compute D^+; Given schema R_i, let D_i denote the restriction of D^+ to R_i
while (**not** *done*) **do**
  if (there is a schema R_i in *result* that is not in 4NF w.r.t. D_i)
    then begin
      let $\alpha \twoheadrightarrow \beta$ be a nontrivial multivalued dependency that holds
      on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \emptyset$;
      result := (*result* $- R_i$) $\cup$ ($R_i - \beta$) $\cup$ (α, β);
    end
    else *done* := true;

Figure 7.16 4NF decomposition algorithm.

Example

- $R = (A, B, C, G, H, I)$

$$F = \{A \twoheadrightarrow B$$
$$B \twoheadrightarrow HI$$
$$CG \twoheadrightarrow H\}$$

- R is not in 4NF since $A \twoheadrightarrow B$ and A is not a superkey for R.
- Decomposition
  1. $R_1 = (A, B)$ $\quad [R_1$ is in 4NF$]$

Example

- $R = (A, B, C, G, H, I)$

$$F = \{A \twoheadrightarrow B$$
$$B \twoheadrightarrow HI$$
$$CG \twoheadrightarrow H\}$$

- R is not in 4NF since $A \twoheadrightarrow B$ and A is not a superkey for R.
- Decomposition
  1. $R_1 = (A, B)$ $[R_1$ is in 4NF$]$
  2. $R_2 = (A, C, G, H, I)$ $[R_2$ is not in 4NF, decompose into R_3 and $R_4]$

Example

- $R = (A, B, C, G, H, I)$

$$F = \{A \twoheadrightarrow B$$
$$B \twoheadrightarrow HI$$
$$CG \twoheadrightarrow H\}$$

- R is not in 4NF since $A \twoheadrightarrow B$ and A is not a superkey for R.
- Decomposition
  1. $R_1 = (A, B)$ $[R_1$ is in 4NF$]$
  2. $R_2 = (A, C, G, H, I)$ $[R_2$ is not in 4NF, decompose into R_3 and $R_4]$
  3. $R_3 = (C, G, H)$ $[R_3$ is in 4NF$]$

Example

- $R = (A, B, C, G, H, I)$

$$F = \{A \twoheadrightarrow B$$
$$B \twoheadrightarrow HI$$
$$CG \twoheadrightarrow H\}$$

- R is not in 4NF since $A \twoheadrightarrow B$ and A is not a superkey for R.
- Decomposition
  1. $R_1 = (A, B)$ $[R_1$ is in 4NF$]$
  2. $R_2 = (A, C, G, H, I)$ $[R_2$ is not in 4NF, decompose into R_3 and $R_4]$
  3. $R_3 = (C, G, H)$ $[R_3$ is in 4NF$]$
  4. $R_4 = (A, C, G, I)$ $[R_4$ is not in 4NF, decompose into R_5 and $R_6]$

Example

- $R = (A, B, C, G, H, I)$

$$F = \{A \twoheadrightarrow B$$
$$B \twoheadrightarrow HI$$
$$CG \twoheadrightarrow H\}$$

- R is not in 4NF since $A \twoheadrightarrow B$ and A is not a superkey for R.
- Decomposition
  1. $R_1 = (A, B)$ $[R_1$ is in 4NF$]$
  2. $R_2 = (A, C, G, H, I)$ $[R_2$ is not in 4NF, decompose into R_3 and $R_4]$
  3. $R_3 = (C, G, H)$ $[R_3$ is in 4NF$]$
  4. $R_4 = (A, C, G, I)$ $[R_4$ is not in 4NF, decompose into R_5 and $R_6]$
     - $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI \longrightarrow A \twoheadrightarrow HI$, (MVD transitivity),
     - and hence $A \twoheadrightarrow I$ (MVD restriction to R_4).

Example

- $R = (A, B, C, G, H, I)$

$$F = \{A \twoheadrightarrow B$$
$$B \twoheadrightarrow HI$$
$$CG \twoheadrightarrow H\}$$

- R is not in 4NF since $A \twoheadrightarrow B$ and A is not a superkey for R.
- Decomposition
  1. $R_1 = (A, B)$ $[R_1$ is in 4NF$]$
  2. $R_2 = (A, C, G, H, I)$ $[R_2$ is not in 4NF, decompose into R_3 and $R_4]$
  3. $R_3 = (C, G, H)$ $[R_3$ is in 4NF$]$
  4. $R_4 = (A, C, G, I)$ $[R_4$ is not in 4NF, decompose into R_5 and $R_6]$
     - $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI \longrightarrow A \twoheadrightarrow HI$, (MVD transitivity),
     - and hence $A \twoheadrightarrow I$ (MVD restriction to R_4).
  5. $R_5 = (A, I)$ $[R_5$ is in 4NF$]$

Example

- $R = (A, B, C, G, H, I)$

$$F = \{A \twoheadrightarrow B$$
$$B \twoheadrightarrow HI$$
$$CG \twoheadrightarrow H\}$$

- R is not in 4NF since $A \twoheadrightarrow B$ and A is not a superkey for R.
- Decomposition
  1. $R_1 = (A, B)$ $[R_1$ is in 4NF$]$
  2. $R_2 = (A, C, G, H, I)$ $[R_2$ is not in 4NF, decompose into R_3 and $R_4]$
  3. $R_3 = (C, G, H)$ $[R_3$ is in 4NF$]$
  4. $R_4 = (A, C, G, I)$ $[R_4$ is not in 4NF, decompose into R_5 and $R_6]$
     - $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI \longrightarrow A \twoheadrightarrow HI$, (MVD transitivity),
     - and hence $A \twoheadrightarrow I$ (MVD restriction to R_4).
  5. $R_5 = (A, I)$ $[R_5$ is in 4NF$]$
  6. $R_6 = (A, C, G)$ $[R_6$ is in 4NF$]$

Further Normal Forms

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

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

Overall Database Design Process

We have assumed schema R is given:

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

E-R Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity:
  - Example: an `employee` entity with:
    - attributes:
      department_name and *building*,
    - functional dependency:
      $department\_name \rightarrow building$,
    - 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:

$$course \bowtie prereq$$

  - 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:
  Instead of `earnings(company_id, year, amount)`, use
  - `earnings_2004`, `earnings_2005`, `earnings_2006`, etc., all on the schema `(company_id, earnings)`.
    - Above are in BCNF, but make querying across years difficult and needs new table each year.
  - `company_year (company_id, earnings_2004, earnings_2005, earnings_2006)`
    - 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

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

Modeling Temporal Data

- **Temporal data** have an association time interval during which the data are *valid*.
- A **snapshot** is the value of the data at a particular point in time.
- Several proposals to extend E-R model by adding valid time to:
  - attributes, e.g., address of an instructor at different points in time.
  - entities, e.g., time duration when a student entity exists.
  - relationships, e.g., time during which an instructor was associated with a student as an advisor.
- But no accepted standard...
- Adding a temporal component results in functional dependencies like:

$$ID \rightarrow street, city$$

  not holding, because the address varies over time.
- A temporal functional dependency $X \rightarrow Y$ holds on schema R if the functional dependency $X \longrightarrow Y$ holds on *all* snapshots for *all* legal instances $r(R)$.

Modeling Temporal Data (Cont.)

- In practice, database designers may add start and end time attributes to relations...
  - E.g.,
    `course(course_id, course_title)`
    is replaced by
    `course(course_id, course_title, start, end)`
  - Constraint: no two tuples can have overlapping valid times,
  - Hard to enforce efficiently.
- Foreign key references may be to current version of data, or to data at a point in time...
  - E.g., student transcript should refer to course information at the time the course was taken.

Plan

Features of Good Relational Design

Functional Dependencies

Decomposition Using Functional Dependencies

Normal Forms

Functional Dependency Theory

Algorithms for Decomposition using Functional Dependencies

Decomposition Using Multivalued Dependencies

More Normal Form

Database-Design Process

Modeling Temporal Data

Atomic Domains and First Normal Form

First Normal Form

- Domain is **atomic** if its elements are considered to be indivisible units...
  - Examples of non-atomic domains:
    - Set of names, composite attributes.
    - Identification numbers like CS101 that can be broken up into parts.
- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic.
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data.
  - Example: Set of accounts stored with each customer, and set of owners stored with each account.
  - We assume all relations are in first normal form (and revisit this in Chapter 22: Object Based Databases).

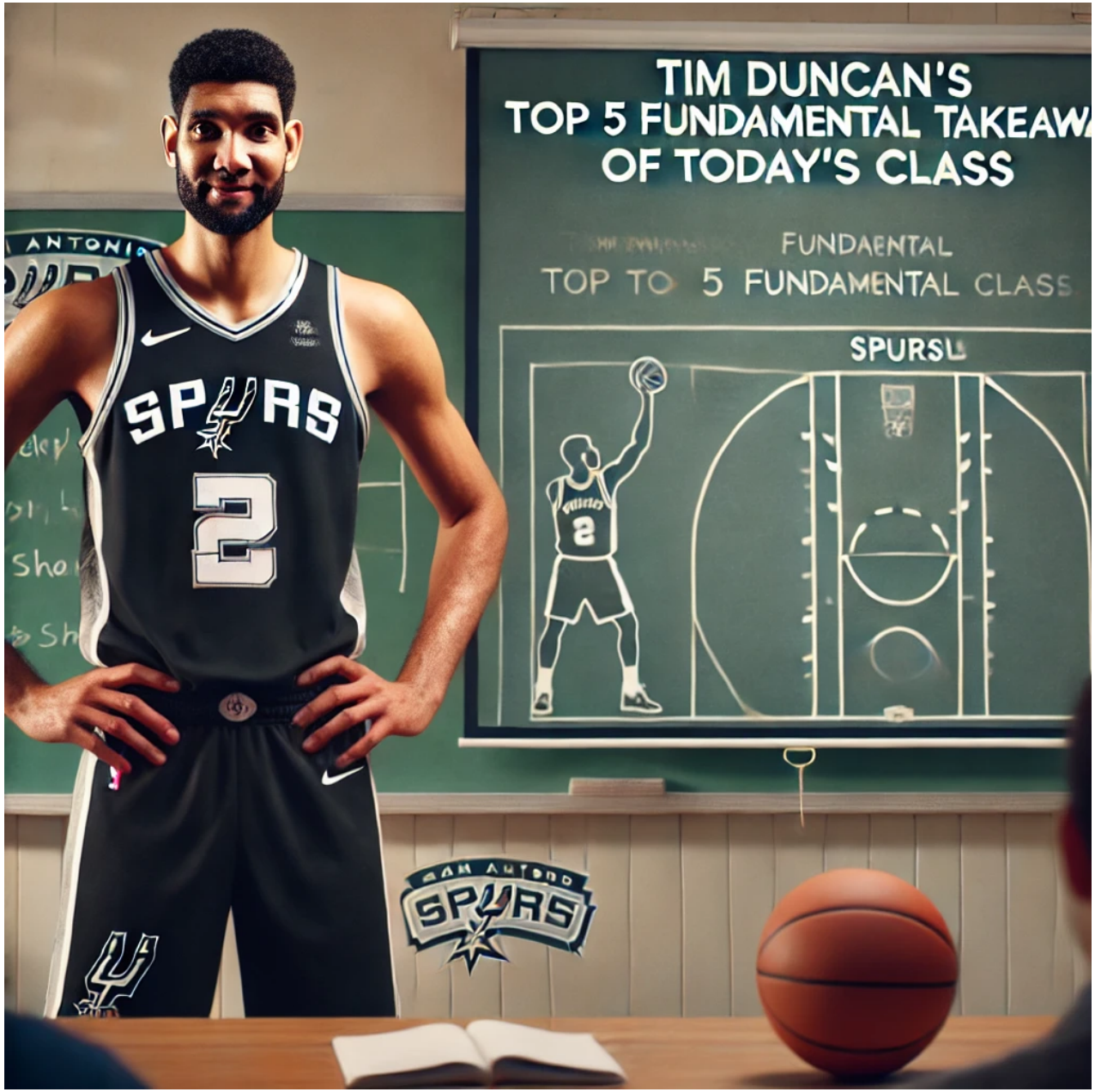
First Normal Form (Cont.)

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.

End of Chapter 7.

TDT5FTOTTC


TIM DUNCAN'S
TOP 5 FUNDAMENTAL TAKEAW
OF TODAY'S CLASS
FUNDAENTAL
TOP TO 5 FUNDAMENTAL CLASS
SPURS
SPURS
SPURS

Top 5 Fundamental Takeaways

Top 5 Fundamental Takeaways

5 Good relational design reduces redundancy and avoids update anomalies by ensuring data is stored without unnecessary repetition or nulls.

Top 5 Fundamental Takeaways

5 Good relational design reduces redundancy and avoids update anomalies by ensuring data is stored without unnecessary repetition or nulls.

4 Lossless and dependency-preserving decomposition ensures that a schema can be split without losing data or making constraint enforcement inefficient.

Top 5 Fundamental Takeaways

5 Good relational design reduces redundancy and avoids update anomalies by ensuring data is stored without unnecessary repetition or nulls.

4 Lossless and dependency-preserving decomposition ensures that a schema can be split without losing data or making constraint enforcement inefficient.

3 Functional dependencies and normal forms guide how to structure schemas to eliminate redundancy while preserving meaningful data relationships.

Top 5 Fundamental Takeaways

5 Good relational design reduces redundancy and avoids update anomalies by ensuring data is stored without unnecessary repetition or nulls.

4 Lossless and dependency-preserving decomposition ensures that a schema can be split without losing data or making constraint enforcement inefficient.

3 Functional dependencies and normal forms guide how to structure schemas to eliminate redundancy while preserving meaningful data relationships.

2 Canonical covers provide a minimal, simplified set of functional dependencies that retain the original semantics for efficient constraint checking.

Top 5 Fundamental Takeaways

5 Good relational design reduces redundancy and avoids update anomalies by ensuring data is stored without unnecessary repetition or nulls.

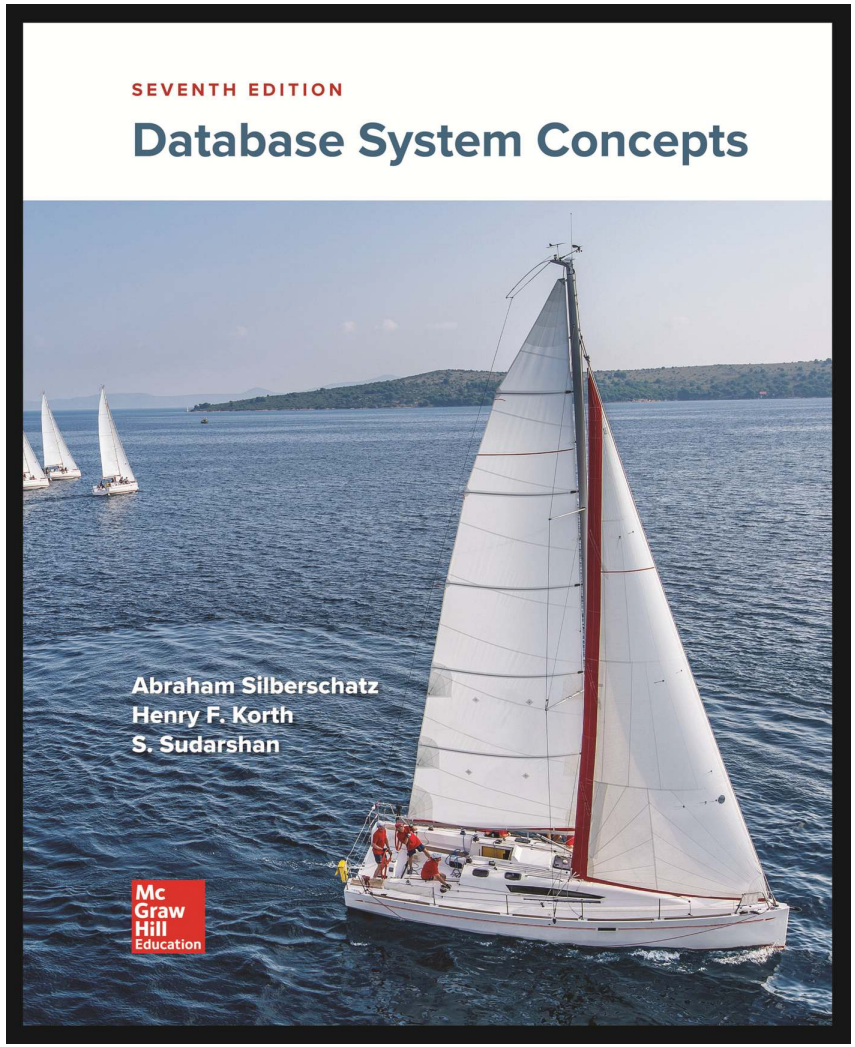
4 Lossless and dependency-preserving decomposition ensures that a schema can be split without losing data or making constraint enforcement inefficient.

3 Functional dependencies and normal forms guide how to structure schemas to eliminate redundancy while preserving meaningful data relationships.

2 Canonical covers provide a minimal, simplified set of functional dependencies that retain the original semantics for efficient constraint checking.

1 Fourth Normal Form (4NF) extends normalization to eliminate redundancy caused by multivalued dependencies, ensuring cleaner data representation.

Database System Concepts



Content has been extracted from *Database System Concepts*, Seventh Edition, by Silberschatz, Korth and Sudarshan. Mc Graw Hill Education. 2019.
Visit `https://db-book.com/`.