Normalization

1 2 3 BCNF Normal Forms

Schema Refinement

- Redundancy
- Schema Refinement
  - Minimizing Redundancy
  - Functional Dependencies (FDs)
  - Normalization using FDs
    - First Normal Form (1NF)
    - Second Normal Form (2NF)
    - Third Normal Form (3NF)
    - Boyce-Codd Normal Form (BCNF)
  - Multivalued Dependencies (MVDs), Join Dependencies JDs)
  - Normalization using MVDs and JDs
    - Higher Normal Forms (4NF, 5NF)

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.

1NF Summarized

- Each attribute must be atomic (single value)
  - No repeating columns within a row (composite attributes)
  - No multi-valued columns.

  All key attributes defined

  All attributes dependent on primary key

- 1NF simplifies attributes
  - Queries become easier.

Dependencies and Normal Form

- The schema R is in 1NF if domain of all attributes of R is atomic
- Desirable
  - PK dependencies
- Undesirable
  - Partial Key : based on part of composite PK
  - Transitive: one nonprime attribute depends on another nonprime attribute

Normal Forms

- Scalar Values
  - 1 NF
- Normal Forms based on PK
  - 2 NF
  - 3 NF
- Normal Forms based on CKs
  - Boyce-Codd Normal Form (BCNF)
- Other Normal Forms
  - 4 NF (Multivalued Dependencies)
  - 5 NF (Join Dependencies)
  - Deal with very rare practical situations

Normal Forms

- 1NF : A relation is in a first normal form, if every tuple contains exactly one value for each attribute
- Example: Suppliers1 (S# , p#, status, city, qty)
- Primary key (S#, P#)
- City→ status (FD)
- FD Diagram (FDD) can be drawn
- Observe
    - Status depends on the city (FD)
    - City depends only on the supplier number

- Primary Key: (s#,p#) -> qty
- Partial: s#-> city
- Transitive: S#->city, city->status implies s#-> status

Redundancies in Suppliers1

Suppliers1 (**s#** , p#, status, city, qty)

- **Insert:** We cannot insert the fact that a particular supplier is located in the city unless he/she actually supplies some part
- The Suppliers1 may not show a supplier is located in Chennai
- **Delete:** If we delete a sole Suppliers1 tuple for a particular supplier, we also delete that he/she is located in a particular city
- **Update:** to update the city for a particular supplier

Schema Refinement for Suppliers1

- Refined Schema Suppliers1
  - Suppliers2 (S#, status, city)
    - Primary : s#-> city
    - Transitive: s#->city, city->status implies s#->status
  - Suppliers_Parts (S#, P#, qty)
    - Primary: (s#,p#) -> qty
- Insert: We can insert the info that s-5 is located in Pune although s-5 does not currently supply any part
- Delete: We do not lose the info that s-18 is located in Surat
  even if all the s-18 tuples from the suppliers_parts are deleted
- Update: city for a supplier can be updated easily through suppliers2

Second Normal Form 2NF

- A relation is in 2NF only if it is in 1NF and every nonkey attribute is irreducibly dependent on the primary key.

  (here, we are assuming only single candidate (hence primary) key case)
- The original relation ***Suppliers1*** can be converted to the 2NF form by taking projection of it to a set of 2 relations ***suppliers2*** and ***suppliers_parts***

- All relations with single attribute PK are in 2 NF!!
- 2NF applies to relations with composite keys

2 NF

- A relation that is in 1NF & every non-PK attribute is fully dependent on the PK, is said to be in 2 NF

1 NF ***Remove all Partial Dependencies*** → 2 NF

2NF Problems: Suppliers2

Suppliers2 (S#, status, city)

- Primary : s#-> city
- Transitive: s#->city, city->status implies s#->status

- Insert: we cannot insert the fact that a particular city has a particular status unless we have some supplier located in that city
- Delete: if we delete a sole tuple for a particular city we delete the information for the supplier and also the information that a city has a particular status
- Update: the status for a city appears many a times in the suppliers2. The change in the Mumbai status from 80 to 90 may need changes in 100 tuples

Schema Refinement for Suppliers2

- Replace Suppliers2 (s#,city, status) by
  - Suppliers3 (S#, city)
  - City_Info (city, status)
- **3NF :** A relation is in a 3NF if it is in 2NF and every nonkey attribute is notransitively dependent on the primary key
- A relation is in a 3NF if nonkey attributes are:
  - Mutually independent
  - Irreducibly/nontransitively dependent on the primary key
- A nonkey attribute is any attribute that does not participate in the primary key of that relation

3 NF

- A relation that is in 1NF & 2 NF & no non-PK attribute is transitively dependent on the PK, is said to be in 3 NF

2 NF —*Remove all Transitive Dependencies*→ **3 NF**

Heath's Theorem

- R(A,B,C) where A B C are sets of attributes
- If R has FD A --> B, then R equals join of {A B} and {A C}

- Example:

S (s#, status, city) ;

PK: s# implies s# --> city and s# --> status

FD: City --> status is troublesome (out of non-candidate key)

Then decomposition

(s#,city), (status, city) is **Dependency Preserving**

Heath's Theorem

- it says that a relation *R* over an attribute set *U* and satisfying a functional dependency $X \rightarrow Y$ can be safely split in two relations having the lossless-join decomposition property, namely into

$$\pi_{XY}(R) \bowtie \pi_{XZ}(R) = R$$

- where $Z = U - XY$ are the rest of the attributes.
- functional dependency provides a simple way to construct a lossless-join decomposition of *R* in two smaller relations

Normal Forms

- Returning to the issue of schema refinement, the first question to be asked is whether any refinement is needed!
- If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help.
- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC.
    - No FDs hold: There is no redundancy here.
    - Given A $\rightarrow$ B: Several tuples could have the same A value, and if so, they'll all have the same B value!

Boyce-Codd Normal Form BCNF (1974)

- 2 or more candidate keys
- Composite candidate keys
- Overlapped keys/ non overlapped keys

- A relation is in BCNF if and only if only determinants are candidate keys

BCNF

- Based on FDs that take into account all candidate keys of a relation
- For a relation with only 1 CK (PK), 3NF & BCNF are equivalent
- A relation is said to be in BCNF if every determinant is a CK

BCNF

- Relation Suppliers1 (**s#,** status, city, **p#,** qty) is not in BCNF. It has 3 determinants s#, city and (s#,p#)
- Primary Key: (s#,p#) -> qty
- Partial: s#-> city
- Transitive: S#->city, city->status implies s#-> status

- Relation Suppliers2(s#, status, city) is not in BCNF. It has s# and city as determinants where city is nonPK attribute

- But SP (s#,p#,qty), SC (s# ,city) and CS (city,status) are in BCNF

Nonoverlapping candidate keys

S(s#, sname,city ,status)

- s# and sname are candidate keys and city->status no longer holds

CK:

- s#-> sname, city, status
- sname-> s#, city, status
- s# <-> sname
- S is in BCNF

Overlapping candidate keys

SSP(s#, sname, p#, qty) is not in BCNF

CK:

- (s#,p#)->qty
- (sname,p#)-> qty
- s# <-> sname

- Redundancies: same (s#, sname) pair repeated for various parts that the supplier is supplying
- SS(s#, sname) , SP(s#,p#,qty)

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 *R* that is not in BCNF. Let $\alpha \rightarrow \beta$ *(nontrivial where* α is not a superkey*)* be the FD that causes a violation of BCNF.
- We decompose *R* into:
  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$
- In our example of *in_dep*,
  - α = *dept_name*
  - β = *building, budget*

  and *in_dep* is replaced by
  - $(\alpha \cup \beta)$ = (*dept_name, building, budget*)
  - $(R - (\beta - \alpha))$ = (*ID, name, dept_name, salary*)

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B)$, $R_2 = (B, C)$
  - Lossless-join decomposition:

    $$R_1 \cap R_2 = \{B\} \quad \text{and } B \rightarrow BC$$

  - Dependency preserving
- $R_1 = (A, B)$, $R_2 = (A, C)$
  - Lossless-join decomposition:

    $$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

BCNF and Dependency Preservation

- database consistency constraints: primary-key constraints, functional dependencies, check constraints, assertions, and triggers.

- 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:

  $i\_ID \rightarrow dept\_name$

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

Dependency Preservation

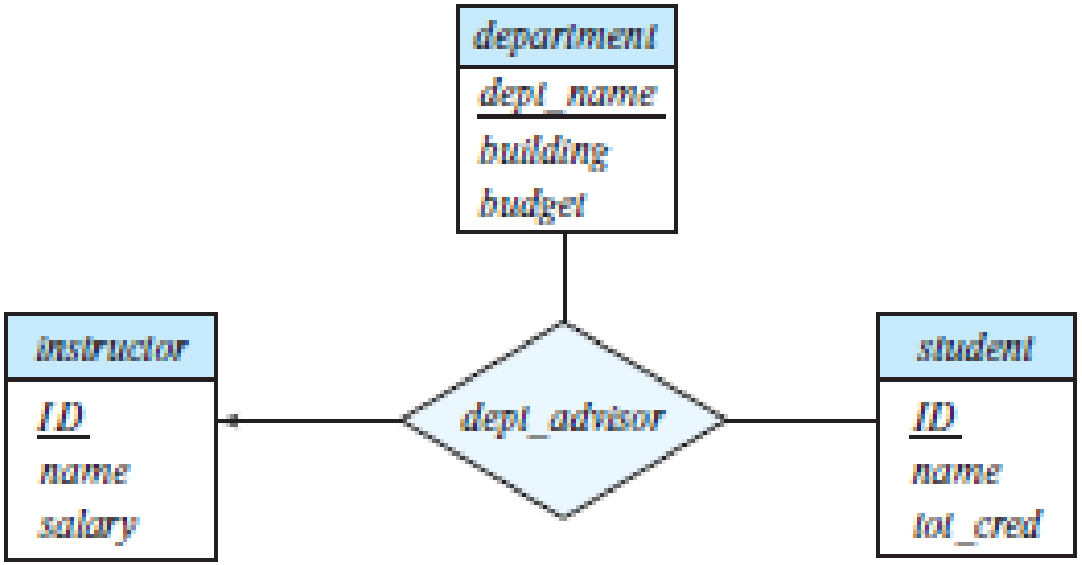

Figure 7.6 The *dept_advisor* relationship set.

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 for *R.*

    (**NOTE***:* each attribute may be in a different candidate key)

- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

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* → *L*, *L* → *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, need to decompose it into a set of relation scheme $\{R_1, R_2, ..., R_n\}$ such that:
  - Each relation scheme is in good form
  - The decomposition is a lossless 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 sugge[illegible]her normal forms, such as Fourth Normal Form (4NF), which we shall see later

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.
- 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 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, 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
    $\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$
    can be shown to hold on R_i, and R_i violates BCNF.
  - We use above dependency to decompose R_i

BCNF Decomposition Algorithm

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

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*)
- Functional dependencies:
  - *course_id*→ *title*, *dept_name*, *credits*
  - *building*, *room_number*→*capacity*
  - *course_id*, *sec_id*, *semester*, *year*→*building*, *room_number*, *time_slot_id*
- A candidate key {*course_id*, *sec_id*, *semester*, *year*}.
- BCNF Decomposition:
  - *course_id*→ *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*)

BCNF Decomposition (Cont.)

- *course* is in BCNF
  - How do we know this?
- *building*, *room_number*→*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*)
- *classroom* and *section* are in BCNF.

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 **do**
  if none of the schemas R_j, $1 \leq j \leq i$ contains $\alpha \beta$
    then begin
      $i := i + 1$;
      $R_i := \alpha \beta$
    end
if none of the schemas R_j, $1 \leq j \leq i$ contains a candidate key for R
  then begin
    $i := i + 1$;
    R_i := any candidate key for R;
  end
/* Optionally, remove redundant relations */
repeat
if any schema R_j is contained in another schema R_k
  then ***/** delete R_j ***/***
    $R_j = R;;$
    $i=i-1;$
return $(R_1, R_2, ..., R_i)$

3NF Decomposition Algorithm (Cont.)

Above algorithm ensures

- Each relation schema R_i is in 3NF
- Decomposition is dependency preserving and lossless-join
- Proof of correctness is at end of this presentation (click here)

3NF Decomposition: An Example

- Relation schema:

  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 1st dependency
  - No other attribute is extraneous, so we get F_C =

    customer_id, employee_id $\rightarrow$ *type*
    employee_id $\rightarrow$ *branch_name*
    customer_id, branch_name $\rightarrow$ *employee_id*

3NF Decompsition Example (Cont.)

- The **for** loop generates following 3NF schema:

  (*customer_id, employee_id, type*)

  (*employee_id, branch_name*)

  (*customer_id, branch_name, employee_id)*

  - Observe that (*customer_id, employee_id, type*) contains a candidate key of the original schema, so no further relation schema needs 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.