RELATIONAL DATABASE DESIGN

Features of Good Relational Design

Schema for the university database.

1. *classroom*(*building, room number,* *capacity*)
2. *department*(*dept_name,* *building*, *budget*)
3. *course*(*course id,* *title*, *dept_name*, *credits*)
4. *instructor*(*ID,* *name*, *dept_name*, *salary*)
5. *section*(*course id, sec id, semester, year,* *building*, *room number*, *time slot id*)
6. *teaches*(*ID, course id, sec id, semester, year*)
7. *student*(*ID,* *name*, *dept_name*, *tot cred*)
8. *takes*(*ID, course id, sec id, semester, year,* *grade*)
9. *advisor*(*s ID, i ID*)
10. *time slot*(*time slot id, day, start time*, *end time*)
11. *prereq*(*course id, prereq id*)

Design Alternatives: Larger-Schema

- Suppose we combine *instructor* and *department* into *inst_dept*
  - *(No connection to relationship set inst_dept)*

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

- Result is possible repetition of information

A Combined Schema without Repetition

- 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

Design Alternatives: Smaller-Schema

- Suppose we had started with *inst_dept.* How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule "if there were a schema (*dept_name, building, budget*), then *dept_name* would be a candidate key"
- Denote as a **functional dependency**:
  - *dept_name* $\rightarrow$ *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

- o *employee1* (*ID*, *name*)
- o *employee2* (*name*, *street, city, salary*)

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

A Lossy Decomposition: It is Loss of information via a bad decomposition.

ID	*name*	*street*	*city*	*salary*
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

employee

ID	*name*
⋮	
57766	Kim
98776	Kim
⋮	

name	*street*	*city*	*salary*
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

natural join

ID	*name*	*street*	*city*	*salary*
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

Our decomposition is unable to represent certain important facts of University employees. So we need to avoid this, and it is said to be **lossy decomposition** and on the other hand if we are able to get by original data then it is called as a **lossless decomposition**.

Example of **Lossless join decomposition**

- Decomposition of $R = (A, B, C)$ *into* $R_1 = (A, B)$ $R_2 = (B, C)$

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

Functional Dependencies

➢ Constraints on the set of legal relations.

➢ 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.*

Definition: Functional dependency is a relationship that exists when one attribute uniquely determines another attribute. It is a relationship between attributes of a table dependent on each other. It helps in preventing data redundancy and gets to know about bad designs**.** Functional dependencies are constraints on the set of legal relations.

Syntax: A -> B

Ex: Account no -> Balance for account table.

Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$, then the functional dependency $\alpha \rightarrow \beta$ holds on R if and only if for any legal relations r(R), whenever any two tuples t1 and t2 of r agree on the attributes of α, they also agree on the attributes of β. That is,

$$t1[\alpha] = t2[\alpha] \Rightarrow t1[\beta] = t2[\beta]$$

Example: Consider *r*(A,*B*) with the following instance of *r.*

A	B
1	4
1	5
3	7

On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

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

  inst_dept (ID, name, salary, dept_name, building, budget).

  We expect these functional dependencies to hold:

  dept_name $\rightarrow$ building and

  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.*
- 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.*

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*.

Partial Dependency: If proper subset of candidate key determines non-prime attribute, it is called partial dependency.

Transitive Dependency: When an indirect relationship causes functional dependency it is called Transitive Dependency.

If P -> Q and Q -> R is true, then P-> R is a transitive dependency.

Trivial Dependency: If a functional dependency (FD) X $\rightarrow$ Y holds, where Y is a subset of X, then it is called a trivial FD. Trivial FDs always hold.

Non-Trivial Dependency: If an FD X $\rightarrow$ Y holds, where Y is not a subset of X, then it is called a non-trivial FD.

Normalization of Database

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables. Normalization is used for mainly two purposes.

- Eliminating redundant (useless) data.
- Ensuring data dependencies make sense i.e data is logically stored to maintain data consistency.

Problem without Normalization: Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not normalized.

- **Updation Anamoly:** We may want to update data in record, but it may exist in different tables or at different places due to redundancy.
- **Insertion Anamoly:** We may try to insert a new record but data may not be fully available or that record itself doesn't exist. Eg: to insert student details without knowing his department leads to Insertion Anamoly.
- **Deletion Anamoly:** When we try to delete a record it might have been saved or exists in some other place of database due to redundancy.

Normalization helps to remove anomalies and ensure that database is in consistent state.

Normalization Types:

Normalization types are divided into following normal forms.

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal Form

First normal form (1NF): A relation is in first normal form if every attribute in every row can contain an atomic (only one single) value. An attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Students

FirstName	LastName	Knowledge
Thomas	Mueller	Java, C++, PHP
Ursula	Meier	PHP, Java
Igor	Mueller	C++, Java

Startsituation

Result after Normalisation

Students

FirstName	LastName	Knowledge
Thomas	Mueller	C++
Thomas	Mueller	PHP
Thomas	Mueller	Java
Ursula	Meier	Java
Ursula	Meier	PHP
Igor	Mueller	Java
Igor	Mueller	C++

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 storage of data.

– ***We assume all relations are in first normal form***

Second normal form (2NF):

A database is in second normal form if it satisfies the following conditions:

- It is in first normal form
- All non-prime attributes are fully functional dependent on the primary key

An attribute that is not part of any candidate key is known as non-prime attribute.

A table that is in 1st normal form and contains only a single key as the primary key is automatically in 2nd normal form.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}

Non prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:

teacher_details table:

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

<u>Third Normal Form(3NF):</u> A relation schema R is in **third normal form** (3NF) if for all:
$\alpha \rightarrow \beta$ in $F+$ with at least one of the following holds:

1. $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
2. α is a superkey for R
3. 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). And, Third condition is a minimal relaxation of BCNF to ensure dependency preservation.

<u>Boyce-Codd Normal Form(BCNF):</u>

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:

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

Example schema *not* in BCNF: *instr_dept* (*<u>ID,</u> name, salary<u>, dept_name,</u> building, budget*)

because *dept_name*$\rightarrow$ *building, budget* holds on *instr_dept,* but *dept_name* is not a super key

<u>Decomposing a Schema into BCNF</u>

Suppose we have a schema R and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.
We decompose R into:

1. (α U β)
2. (R - (β - α))

In our example,

α = *dept_name*
β = *building, budget*

and *inst_dept* is replaced by

- (α U β) = (*dept_name, building, budget*)
- (R - (β - α)) = (*ID, name, salary, dept_name*)

<u>BCNF and Dependency Preservation</u>

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

<u>How good is BCNF?</u>

There are database schemas in BCNF that do not seem to be sufficiently normalized

Ex:1 Consider a relation
inst_info (ID, child_name, phone)

where an instructor may have more than one phone and can have multiple children

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

- 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)

- Therefore, it is better to decompose *inst_info* into:

inst_child

ID	*child_name*
99999	David
99999	David
99999	William
99999	Willian

inst_phone

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

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF)

Ex: Consider a database: *classes* (*course, teacher, book*) such that (*c, t, b*) ∈ *classes* means that *t* is qualified to teach *c,*and *b* is a required textbook for *c*
The database is supposed to list for each course the set of teachers any one of which can be the course's instructor, and the set of books, all of which are required for the course (no matter who teaches it).

course	Teacher	book
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Stallings
operating systems	Pete	OS Concepts
operating systems	pete	Stallings

Classes

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

Therefore, it is better to decompose *classes* into:
Teaches:

Course	teacher
database	Avi
database	Hunk
database	Sudarshan
operating systems	Avi
operating systems	Jim

Text:

Course	Book
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF),

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 pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] = t_1[\beta]$$
$$t_3[R - \beta] = t_2[R - \beta]$$
$$t_4[\beta] = t_2[\beta]$$
$$t_4[R - \beta] = t_1[R - \beta]$$

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 $r(R)$
  $< y_1, z_1, w_1 > \in r$ and $< y_1, z_2, w_2 > \in r$
  Then $< y_1, z_1, w_2 > \in r$ and $< y_1, z_2, w_1 > \in r$
- Note that since the behavior of Z and W are identical it follows that
  $Y \rightarrow\rightarrow Z$ if $Y \rightarrow\rightarrow W$
- In our example:
  ID $\rightarrow\rightarrow$ *child_name*
  ID $\rightarrow\rightarrow$ *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 $Y \rightarrow\rightarrow Z$

Fourth Normal Form (4NF): A relation schema R is in **4NF** with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:

  - $\alpha \rightarrow\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
  - α is a superkey for schema R
- If a relation is in 4NF it is in BCNF

Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
  - lead to **project-join normal form (PJNF)** (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used

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 $\{R1, R2, ..., Rn\}$ such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving.

Functional-Dependency Theory

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

For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$

The set of all functional dependencies logically implied by F is the ***closure*** of F.
We denote the *closure* of F by $F+$.

We can find all of F+ by applying **Armstrong's Axioms**:

1. if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
2. if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ **(augmentation)**
3. if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**

These rules are
- **sound** (generate only functional dependencies that actually hold) and
- **complete** (generate all functional dependencies that hold).

Example:

Let R = (A, B, C, G, H, I) and fd are given by
$F = \{ A \rightarrow B$
$A \rightarrow C$
$CG \rightarrow H$
$CG \rightarrow I$
$B \rightarrow H\}$

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

and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity.

We can further simplify manual computation of $F+$ by using the following additional rules.
1. If $\alpha \rightarrow \beta$ holds *a*nd $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds **(union)**
2. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition)**
3. If $\alpha \rightarrow \beta$ holds *a*nd $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds **(pseudotransitivity)**

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

Algorithm 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 f1and f2 in F +
            if f1 and f2 can be combined using transitivity
               then add the resulting functional dependency to F +
until F + does not change any further
```

Closure of Attribute Sets: Given a set of attributes a, define the *closure* of a under *F* (denoted by a+) as the set of attributes that are functionally determined by a under *F*

<u>Algorithm to compute a+, the closure of a under *F*</u>

```
        result := a;
        while (changes to result) do
            for each β → γ in F do
                begin
                    if β ⊆ result then  result := result ∪ γ
                end
```

Example of Attribute Set Closure

Let R = (A, B, C, G, H, I) and fd's are given by

$F = \{A \rightarrow B$
$A \rightarrow C$
$CG \rightarrow H$
$CG \rightarrow I$
$B \rightarrow H\}$

Now to compute *(AG)+* we trace above algorithm to get :

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. Does $AG \rightarrow R$? == Is (AG)+ $\supseteq$ R
2. Is any subset of AG a superkey?
    1. Does $A \rightarrow R$? == Is (A)+ $\supseteq$ R
    2. Does $G \rightarrow R$? == Is (G)+ $\supseteq$ R

Uses of Attribute Closure:

There are several uses of the attribute closure algorithm:

1. Testing for super key:
    a. To test if α is a superkey, we compute $\alpha+$, and check if $\alpha+$ contains all attributes of *R*.
2. Testing functional dependencies
    a. To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in *F+*), just check if $\beta \subseteq \alpha+$.
    b. That is, we compute $\alpha+$ by using attribute closure, and then check if it contains β.
    c. Is a simple and cheap test, and very useful
3. Computing closure of F
    a. For each $\gamma \subseteq R$, we find the closure $\gamma+$, and for each $S \subseteq \gamma+$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover:
Sets of functional dependencies may have redundant dependencies that can be inferred from the others.

For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C\}$

Parts of a functional dependency may be redundant

Ex:: on RHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow CD\}$ can be simplified to
$\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$

Ex:: on LHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad AC \rightarrow D\}$ can be simplified to
$\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$

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.

Canonical Cover Definition: A *canonical cover* for *F* is a set of dependencies *Fc* such that

1. *F* logically implies all dependencies in *Fc,* and
2. *Fc* logically implies all dependencies in *F,* and
3. No functional dependency in *Fc* contains an extraneous attribute, and
4. Each left side of functional dependency in *Fc* is unique.

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

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

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

Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$
B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (I.e. the result of dropping B from $AB \rightarrow C$).

Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
C is extraneous in $AB \rightarrow CD$ since $AB \rightarrow C$ can be inferred even after deleting C.

Testing if an Attribute is Extraneous: *Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.*

1) *To test if attribute $A \in \alpha$ is extraneous in α*
   a) *compute $(\{\alpha\} - A)+$ using the dependencies in F*
   b) *check that $(\{\alpha\} - A)+$ contains β; if it does, A is extraneous in α*
2) *To test if attribute $A \in \beta$ is extraneous in β*
   a) *compute $\alpha+$ using only the dependencies in*
      $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,
   b) *check that $\alpha+$ contains A; if it does, A is extraneous in β*

Computing a Canonical Cover:

To compute a canonical cover for *F*:

```
repeat
        Use the union rule to replace any dependencies in F
                α1 → β1 and α1 → β2 with α1 → β1 β2
        Find a functional dependency α → β with an
                extraneous attribute either in α or in β
        If an extraneous attribute is found, delete it from α → β
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

Let $R = (A, B, C)$ and
$F = \{A \rightarrow BC$
$B \rightarrow C$
$A \rightarrow B$
$AB \rightarrow C\}$

Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$

1. A is extraneous in $AB \rightarrow C$
   a. Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
      i. Yes: in fact, $B \rightarrow C$ is already present!
   b. Set is now $\{A \rightarrow BC, B \rightarrow C\}$
2. C is extraneous in $A \rightarrow BC$
   a. Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
      i. Yes: using transitivity on $A \rightarrow B$ and $B \rightarrow C$.
         1. Can use attribute closure of A in more complex cases
3. The canonical cover is: $\{ A \rightarrow B , B \rightarrow C \}$

Decomposition using functional dependencies

Lossless-join Decomposition: For the case of $R = (R1, R2)$, we require that for all possible relations r on schema R

$$r = \prod_{R1}(r) \bowtie \prod_{R2}(r)$$

A decomposition of R into $R1$ and $R2$ is lossless join if and only if at least one of the following dependencies is in $F+$:

1. $R1 \cap R2 \rightarrow R1$
2. $R1 \cap R2 \rightarrow R2$

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: *Let R = (A, B, C) and F = {A → B, B → C)*
It Can be decomposed in two different ways

1. *R*1 = *(A, B), R*2 = *(B, C)*
    a. Lossless-join decomposition:
        i. *R*1 ∩ *R*2 = {*B*} and *B* → *BC*
    b. Dependency preserving
2. *R1 = (A, B), R*2 = *(A, C)*
    a. Lossless-join decomposition:
        i. *R*1 ∩ *R*2 = {*A*} and *A* → A*B*
    b. Not dependency preserving
       (cannot check *B* → *C* without computing *R1*   *R*2)

Dependency Preservation: Let *Fi* be the set of dependencies *F* + that include only attributes in *Ri.* Decomposition is dependency preserving, if

$(F1 \cup F2 \cup \ldots \cup Fn)+ = F+$

If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Testing for Dependency Preservation: To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of *R* into *R*1, *R*2, …, *R*n we apply the following test (with attribute closure done with respect to *F*)

result = α
while (changes to *result*) do
  for each *Ri* in the decomposition
    t = (*result* ∩ *Ri*)+ ∩ *Ri*
    result = *result* ∪ *t*

If *result* contains all attributes in β, 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 takes polynomial time, instead of the exponential time required to compute *F*+ and (*F*1 ∪ *F*2 ∪ … ∪ *F*n)+

Example:
Let R = (*A, B, C*) and *F* = {*A* → *B* , *B* → *C*}
Key = {*A*}
R is not in BCNF
Decomposition *R*1 = (*A, B), R*2 = *(B, C)*
  a. *R*1 and *R*2 in BCNF
  b. Lossless-join decomposition
  c. Dependency preserving

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 super key of *R*.

<u>Simplified test</u>: To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F* for violation of BCNF, rather than checking all dependencies in *F*+.

- If none of the dependencies in *F* causes a violation of BCNF, then none of the dependencies in *F*+ will cause a violation of BCNF either.

However, using only *F* is incorrect when testing a relation in a decomposition of R
Consider *R* = (*A, B, C, D, E*), with *F* = { *A* → *B*, *BC* → *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* → *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,

a. Either test Ri for BCNF with respect to the restriction of F to Ri (that is, all FDs in F+ that contain only attributes from Ri)
b. or use the original set of dependencies *F* that hold on *R*, but with the following test:
   1. for every set of attributes $\alpha \subseteq Ri$, check that α+ (the attribute closure of α) either includes no attribute of $Ri- \alpha$, or includes all attributes of *Ri*.

   ii. If the condition is violated by some $\alpha \rightarrow \beta$ in *F*, the dependency

   $\alpha \rightarrow (\alpha+ - \alpha) \cap Ri$

   can be shown to hold on *Ri*, and *Ri* violates BCNF.

   iii. We use above dependency to decompose *Ri*

BCNF and Dependency Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving
Let R = (*J, K, L*) and *F* = {*JK* $\rightarrow$ *L,* *L* $\rightarrow$ *K* }
Two candidate keys = *JK* and *JL*

1. *R* is not in BCNF
2. Any decomposition of *R* will fail to preserve
   1. *JK* $\rightarrow$ *L*

This implies that testing for *JK* $\rightarrow$ *L* requires a join

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

Consider Relation R: let *R* = (*J, K, L*) and *F* = {*JK* $\rightarrow$ *L, L* $\rightarrow$ *K* }

- Two candidate keys: *JK* and *JL*
- *R* is in 3NF
  - *JK* $\rightarrow$ *L JK* is a superkey
    L $\rightarrow$ *K* *K* is contained in a candidate key

- Relation *dept_advisor*:
  - *dept_advisor* (*s_ID, i_ID, dept_name)*
    F = {*s_ID, dept_name* $\rightarrow$ *i_ID, i_ID* $\rightarrow$ *dept_name*}
  - Two candidate keys: *s_ID, dept_name,* and *i_ID, s_ID*
  - *R* is in 3NF
    - *s_ID, dept_name* $\rightarrow$ *i_ID* *s_ID*
      - *dept_name* is a superkey
    - *i_ID* $\rightarrow$ *dept_name*
      - *dept_name* is contained in a candidate key

Redundancy in 3NF: There is some redundancy in this schema

Example of problems due to redundancy in 3NF

Let R = (J, K, L) and F = {*JK* $\rightarrow$ *L, L* $\rightarrow$ *K* }

J	L	K
J1	L1	K1
J2	L1	K1
J3	L1	K1
null	L2	K2

- repetition of information (Ex:, the relationship *l*1, *k*1)
  - (*i_ID, dept_name)*

- need to use null values (Ex:, to represent the relationship
  *l*2, *k*2 where there is no corresponding value for *J*).
  - (*i_ID, dept_nameI)* if there is no separate relation mapping instructors to departments

Testing for 3NF: Optimization: Need to check only FDs in *F*, need not check all FDs in *F*+.

1. Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a super key.
2. If α is not a super key, we have to verify if each attribute in β is contained in a candidate key of *R*
   a. this test is rather more expensive, since it involve finding candidate keys
   b. testing for 3NF has been shown to be NP-hard
   c. Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

Comparison of BCNF and 3NF: It is always possible to decompose a relation into a set of relations that are in 3NF such that:

1. the decomposition is lossless
2. 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:

1. BCNF.
2. Lossless join.
3. 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 super keys. Can specify FDs using assertions, but they are expensive to test that 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.

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.