Normalisation

Normalization: branch of relational theory providing design insights.

The goals of normalization:

- be able to characterise the level of redundancy in a relational schema
- provide mechanisms for transforming schemas to remove redundancy

Normalization draws heavily on the theory of functional dependencies

Schema Design

Consider the following relation for *BankLoans*:

branchName	branchCity	assets	custName	loanNo	amount
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-15	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
North Town	Rye	3700000	Hayes	L-16	1300

Schema Design

The *BankLoans* relation exhibits update anomalies (insert, update, delete).

The cause of these problems can be stated **in terms of *fds***

- a branch is located in one city   *branchName* $\rightarrow$ *branchCity*
- a branch may handle many loans   *branchName* $\nrightarrow$ *loanNo*

In other words, some attributes are determined by *branchName*, while others are not.

This suggests that we have two separate notions (branch and loan) mixed up in a single relation

Schema Design

To improve the design, decompose the *BankLoans* relation.

The following decomposition is not helpful:

Branch(branchName, branchCity, assets)
CustLoan(custName, loanNo, amount)

because we lose information (which branch is a loan held at?)

Clearly, we need to leave some "connection" between the new relations, so that we can reconstruct the original information if needed.

Another possible decomposition:

BranchCust(branchName, branchCity, assets, custName)
CustLoan(custName, loanNo, amount)

Schema Design

The *BranchCust* relation instance:

branchName	branchCity	assets	custName
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
North Town	Rye	3700000	Hayes

The *CustLoan* relation instance:

custName	loanNo	amount
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-15	1500
Jones	L-93	500
Turner	L-11	900
Hayes	L-16	1300

The result:

- *BranchCust* still has redundancy problems.
- *CustLoan* doesn't, but there is potential confusion over L-15.
- But even worse, when we put these relations back together to try to re-create the original relation, we get some extra tuples! Not good.

The result of
Join(BranchCust,CustLoan) On CustName

branchName	branchCity	assets	custName
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
North Town	Rye	3700000	Hayes

custName	loanNo	amount
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-15	1500
Jones	L-93	500
Turner	L-11	900
Hayes	L-16	1300

branchName	branchCity	assets	custName	loanNo	amount
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	*Brooklyn*	*9000000*	*Jones*	*L-93*	*500*
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	*Horseneck*	*1700000*	*Hayes*	*L-16*	*1300*
Downtown	Brooklyn	9000000	Jackson	L-15	1500
Mianus	Horseneck	400000	Jones	L-93	500
Mianus	*Horseneck*	*400000*	*Jones*	*L-17*	*1000*
Round Hill	Horseneck	8000000	Turner	L-11	900
North Town	Rye	3700000	Hayes	L-16	1300
North Town	*Rye*	*3700000*	*Hayes*	*L-15*	*1500*

Schema Design

This is clearly not a successful decomposition.

The fact that we ended up with extra tuples was symptomatic of losing some critical "connection" information during the decomposition.

Such a decomposition is called a *lossy decomposition*.

In a good decomposition, we should be able to reconstruct the original relation exactly:

if *R* is decomposed into *S* and *T*, then *Join(S,T)* = *R*

Such a decomposition is called *lossless join decomposition*.

How do we decide on this decomposition strategy?

→ we analyse “normal forms” and apply “normalisation process”

Normal Forms

Normalization theory defines six *normal forms* (NFs).

Each normal form:

- defines the properties that a schema must satisfy (e.g., 1NF must have atomic values)
- gives guarantees about presence/absence of update anomalies

Higher normal forms have less redundancy ⇒ less update problems

Normal forms:

- First, Second, Third Normal Forms (1NF, 2NF, 3NF) (Codd 1972)
- Boyce-Codd Normal Form (BCNF) (1974)
- Fourth Normal Form (4NF) (Zaniolo 1976, Fagin 1977)
- Fifth Normal Form (5NF) (Fagin 1979)

NF hierarchy: 5NF ⇒ 4NF ⇒ BCNF ⇒ 3NF ⇒ 2NF ⇒ 1NF

1NF allows most redundancy; 5NF allows least redundancy.

Normal Forms

The use of normal forms:

- First, check if a given relation is in r-NF (where r-NF is your desired level of NF)
- Second, transform the given relation to r-NF

In practice, BCNF and 3NF are the most important.

- these are generally the "acceptable/desired normal forms" for relational design,
- 1NF/2NF (too much redundancy), 4NF/5NF (handles redundancy that are not likely to be common in practice)

<u>Our desired normal forms are:</u>

Boyce-Codd Normal Form (BCNF):

- eliminates all redundancy due to functional dependencies
- but may not preserve original functional dependencies

Third Normal Form (3NF):

- eliminates most (but not all) redundancy due to *fd*s
- guaranteed to preserve all functional dependencies

Relational Decomposition

From given relations, achieving Normal Forms is one of the schema design strategies (i.e., going from possibly redundant relations into redundancy free forms)

The standard transformation technique to remove redundancy:

- ***decompose*** relation *R* into relations *S* and *T*

We accomplish decomposition by

- selecting (overlapping) subsets of attributes
- forming new relations based on attribute subsets
- Properties: $R = S \cup T$, $S \cap T \neq \{\}$ and ideally $r(R) = s(S) \bowtie t(T)$

We may require several decompositions to achieve acceptable NF.

Normalization algorithms tell us how to choose *S* and *T*.

Boyce-Codd Normal Form (BCNF)

A relation schema R is in BCNF w.r.t a set F of functional dependencies iff:

- for all *fd*s $X \rightarrow Y$ in F
  - either $X \rightarrow Y$ is trivial (i.e. $Y \subset X$)
  - or X is a superkey (i.e., contains a key)

In another words: if a $X \rightarrow A$ is a *nontrivial* fd (i.e., $A \notin X$) in R, then X must be a *superkey (i.e., contains a key)*

Some simple "short hands" about BCNF:

- any relation with two-attributes is in BCNF
- any relation with key K, other attributes X, *(i.e.,* $(K \cup X = R)$) and $K \rightarrow X$, is in BCNF

A DB schema is in BCNF if all relation schemas are in BCNF.

Example

Beers(name, manf, manfAddr)

FD's: name $\rightarrow$ manf,   manf $\rightarrow$ manfAddr

The key is {name}

Is the above in BCNF?

Work out:

name $\rightarrow$ manf does not violate BCNF,

manf $\rightarrow$ manfAddr does violate BCNF. manfAddr depends on manf, which does not contain the key

Beers is not in BCNF

Another Example

Drinkers(name, addr, beersLiked, manf, favBeer)

FDs: name $\rightarrow$ addr favBeer,   beersLiked $\rightarrow$ manf

The key is {name, beersLiked}

Is the above in BCNF?

Work out:

- name $\rightarrow$ addr favBeer ... the left side is not a superkey (i.e., does not contains the key)
- beersLiked $\rightarrow$ manf ... the left side is not a superkey (i.e., does not contains the key)
- Drinkers is not in BCNF.

Boyce-Codd Normal Form

If we transform a schema into BCNF, we are guaranteed:

- no update anomalies due to *fd*-based redundancy (especially the ones that are dependent on non key attributes)
- lossless join decomposition

However, we are *not* guaranteed to preserve all *fd*s from the original schema exist in the new schema

- This may be a problem if the *fd*s contain significant semantic information about the problem domain.
- If we need to preserve dependencies, use 3NF.

BCNF Decomposition

The following algorithm converts an arbitrary schema to BCNF:

```
Inputs: schema R, set F of fds
Output: set Res of BCNF schemas

Res = {R};
while (any schema S ∈ Res is not in BCNF) {
    choose an fd X → Y on S that violates BCNF
    Res = (Res-S) ∪ (S-Y) ∪ XY
}
```

The last step:

- Remove S from Res // we are going to replace S with new ones
- Remove {*Y* } the right-hand side, from *S // let's denote new S as S'*
- Make a new table using *{XY}*
- *Now Res contains S' , plus, XY, plus any existing schema in Res*

BCNF Decomposition

Example (the *BankLoans* schema):

BankLoans(branchName, branchCity, assets, custName, loanNo, amount)

FD : {

branchName $\rightarrow$ assets,branchCity,

loanNo $\rightarrow$ amount,branchName

}

The key {branchName, custName, loanNo}

Produce a BCNF decomposition of BankLoans

BCNF Exercise

Consider the schema *R* and set of fds *F*

$R = ABCDEFGH$

$F = \{ ABH \rightarrow C, A \rightarrow DE, BGH \rightarrow F, F \rightarrow ADH, BH \rightarrow GE \}$

$Key = BH$

Produce a BCNF decomposition of *R*.

Third Normal Form — Motivation

There is one structure of FDs that causes trouble in BCNF

R = ABC, FD: AB $\rightarrow$ C and C $\rightarrow$ B

Keys: AB, AC

(e.g., A = street address, B = city, C = zip code)

C $\rightarrow$ B is a BCNF violation,
so we must decompose R(ABC) into

- R(AC), ??
- R(BC), FD: C $\rightarrow$ B

- What happens to the functional dependency AB $\rightarrow$ C ??

An Unenforceable FD

If we decompose ABC into AC and BC then we cannot enforce AB $\rightarrow$ C by checking FDs in the decomposed relations

street	zip
545 Tech Sq.	02138
545 Tech Sq.	02139

city	zip
Cambridge	02138
Cambridge	02139

Join tuples with equal zip codes.

street	city	zip
545 Tech Sq.	Cambridge	02138
545 Tech Sq.	Cambridge	02139

Although no FDs were violated in the decomposed relations, FD street city $\rightarrow$ zip is violated by the database as a whole

3NF Lets Us Avoid This Problem

3rd Normal Form (3NF) modifies the BCNF condition so we do not have to decompose in this problem situation

A relation schema *R* is in 3NF w.r.t a set *F* of functional dependencies iff:

- for all *fd*s $X \rightarrow Y$ in *F*
  - either $X \rightarrow Y$ is trivial (i.e. $Y \subset X$)
  - or *X* is a superkey
  - or *Y* is a single attribute from a key

e.g., R = ABC, FD: AB $\rightarrow$ C and C $\rightarrow$ B, Keys: AB

A DB schema is in 3NF if all relation schemas are in 3NF. The extra condition represents a slight weakening of BCNF requirements

Third Normal Form

If we transform a schema into 3NF, we are guaranteed:

- lossless join decomposition
- the new schema preserves all of the *fd*s from the original schema

However, we are *not* guaranteed:

- no update anomalies due to *fd*-based redundancy

Whether to use BCNF or 3NF depends on overall design considerations.

Third Normal Form

The following algorithm converts an arbitrary schema to 3NF:

```
Inputs: schema $R$, set $F$ of $fds$
Output: set $Res$ of 3NF schemas

let $F_c$ be a minimal cover for $F$
$Res$ = {}
for each $fd$ $X \rightarrow Y$ in $F_c$ {
    if (no schema $S \in Res$ contains $XY$) {
        $Res = Res \cup XY$
    }
}
if (no schema $S \in Res$ contains a candidate key for $R$) {
    $K$ = any candidate key for $R$
    $Res = Res \cup K$
}
```

Third Normal Form

Two critical concepts in the algorithm:

Minimal cover F_c for *F*A set *F* of fds is minimal if

- every fd $X \rightarrow Y$ is *simple*
  (Y is a single attribute, X ->AB = X->A, X->B)
- every fd $X \rightarrow Y$ is *left-reduced*
  (no *redundant attributes on the left side, AB->Y = A->Y (if safe to do so))*
- every fd $X \rightarrow Y$ is *necessary*
  (no $X \rightarrow Y$ can be removed without changing F^+)
- Summary: right-reduce, left-reduce, eliminate redundant fds

Candidate keys

- Minimal superkey (if necessary, use attribute closure to work out a key)

3NF decomposition exercise (1)

Consider the schema R and set of fds F

$R = ABCDEFGH$

$F = F_c = \{ ABH \rightarrow C, A \rightarrow D, C \rightarrow E, F \rightarrow A, E \rightarrow F, BGH \rightarrow E \}$

key = BGH

Produce a 3NF decomposition of R.