Relational DB Design: Functional Dependencies / Normalization

- Schema + Query Language determine application programs
- Goal: Have a schema that make queries simpler, avoids redundancy, update anomalies, loss of info

Example:

Schema 1: Emp (eno, ename, byr, sal, dno)
Dept (dno, dname, floor, mgr) ← eno of mgr

Q1: Find all employee who make more than their manager.

```
Select E.ename
From Emp E, Dept D, Emp E2
where (D.mgr = E2.eno) and (E.dno = D.dno)
    and (E.sal > E2.sal)
```

Q2: Find all ~~emp~~ depts with a max salary > 2X avg sal

```
Select (d.dname)
From Emp e, Dept d
where (d.dno = e.dno)
group by d.dno, d.dname
having max (e.sal) > 2*AVG(e.sal)
```

Schema 2: ED (eno, ename, byr, sal, dno, dname, floor, mgr)

Q1:
```
Select e.ename
from ED e1, ED e2
where (e1.mgr = e2.eno) and (e1.sal > e2.sal)
```

Q2:
```
Select e.dname
From ED e
group by e.dno, e.dname
having MAX(e.sal) > 2*AVG(e.sal)
```

* We get simpler queries, but we also get into trouble:

- Redundancy
  - each dpt is repeated once for each employee
  - there is potential inconsistency (update anomalies).
    We may change the location of a dpt in one
    tuple leaving it with the old value in another
  - A simple change is translated into multiple replacements
    (e.g. change the manager of the "shoe" department)

- No independent existence:
  - A dpt can not exist without employees
  - when we delete the last employee of a dpt, we
    automatically lose track of the dept.

Objective of DB Design

- No redundancy for space efficiency
- update integrity
- Semantic clarity
- Linguistic efficiency (the simpler the queries the better for both the user and query optimizer)
- Performance (binary relations imply most queries will have a large number of joins)

Tools for DB design : Functional Dependencies + Normalization

Functional Dependency : Let A, B be sets of attributes of R.
The FD $A \to B$ holds if a value for A uniquely determines a value for B.

More formally : $A \to B$ if :

$\forall$ pairs (t_1, t_2) of tuples in relation r such that $t_1[A] = t_2[A]$, it is also the case $t_1[B] = t_2[B]$

Note let K be a super-key of R, then

$$K \to R$$

Examples

Emp : eno $\to$ ename   e.sal $\not\to$ e.name

eno $\to$ dno

Dept : dno $\to$ mgr

dno $\to$ floor

ED : dno $\to$ floor

eno $\to$ sal

eno $\to$ mgr

Note, a FD is a property of a particular relation schema, and ~~not~~ not of this instance.

Example

eno	ename	byr	sal	dno
17	Mike	60	40	1
18	Sally	55	85	1
19	Sara	65	50	2
20	Julie	70	19	2
21	John	48	80	1
22	Alex	70	20	3
24	Mike	60	45	1

looks like

ename → byr

ename → dno

but this is not right. For example I could add tuple

23 Mike 49 22 6

ex:

Various types of FD's help identify bad designs

- Trivial Dependency : $(A, B) \rightarrow A$ (Identity)
- Partial dependency : (A, B) is a key and $A \rightarrow C$

Ex: supply ($\underset{\text{supplier \#}}{\text{sno}}$, $\underset{\text{part \#}}{\text{pno}}$, projno, scity, projcity, qty)

key is (sno, pno, projno)

$$\left.\begin{array}{l} sno \rightarrow scity \\ projno \rightarrow projcity \end{array}\right\} \text{partial dependencies}$$

- Transitive Dependency : A is a key, B is not, and $A \rightarrow B \rightarrow C$

Ex : ED(eno, ... , dno, ... , mgr) where eno is the key.

$$\left.\begin{array}{c} eno \rightarrow dno \rightarrow mgr \\ eno \rightarrow mgr \end{array}\right\} \text{transitive dependency}$$

Depending on the existence of various FDs, schemes are classified in various <u>Normal Forms</u>

1NF : Every attribute has an atomic value (as opposed to set value) Note, <u>all</u> relational model schemas are by definition 1NF

(partial dependencies ⇒ redundancy)

2NF : 1NF and no partial dependencies

(transitive dependencies ⇒ redundancy)

3NF : 2NF and no transitive dependencies to attributes that are not part of a key. Equivalently, if $X \to A$ is a FD then either:

a) it is trivial, or

b) X is a superkey, or

c) A is a subset of a candidate key.

BCNF : (Boyce Codd Normal Form) 3NF and no transitive dependencies. Equivalently, if $X \to A$ is a FD then either:

a) it is trivial, or

b) X is a super key

4NF (5, 6 ...) Multivalue dependencies (rarely used, mostly just theory for theories sake)

<u>Distinguishing Example Schemas</u>

1NF but not 2NF: Supply(sno, pno, projno, scity, projcity, qty)

Key = (sno, pno, projno)

2NF but not 3NF: ED(eno, ename, byr, sal, dno, dname, floor, mgr)

Key = (eno)

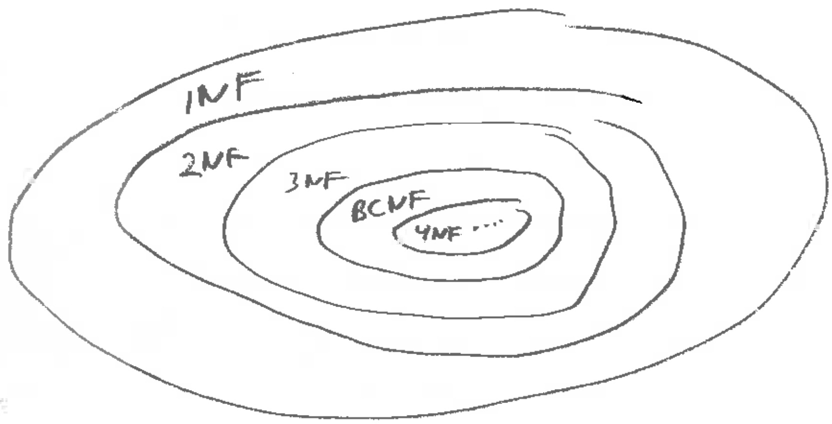
One recipt per person, food pair

3NF but not BCNF : Restaurant (person, food, rec_number)

Key = (person, food)

This is 3NF because it has transitive dependency to an attribute that is part of the key.

Each normal form is included in the next one higher up:



Ideally, a decomposition should satisfy:

- BCNF
- Lossless joins
- dependency preservation

If we can not get this, we settle for

- 3NF
- Lossless joins
- dependency preservation

resident asst id

3NF example   name   floor   age   RAid

Lossless join: The decomposition should be done in a way such that no information is lost.

Ex: Borrow = (bname, lnum, cname, amount)

⇓ decompose

Amt-schema (amount, cname)
Loan-schema (bname, lnum, amount)

loan

bname	lnum	amount
hampton	1	1000
norfolk	2	1500
wmbg	3	2000
NN	4	1500

amt

amount	cname
1000	Sue
1500	Mike
2000	Biff
1500	Jane

amt ⋈ loan

=

hampton	1	1000	Sue
norfolk	2	1500	Mike
norfolk	2	1500	Jane
wmbg	3	2000	Biff
NN	4	1500	Mike
NN	4	1500	Jane

wrong

Gives us incorrect results! Hence we have lost information.

19.89

Let $F \triangleq$ set of semantically obvious FD

More FD's can be inferred from F

$F^+ \triangleq$ set of all inferred FDs

Example

$$F = \begin{Bmatrix} A \to B \\ A \to C \\ B \to K \end{Bmatrix} \quad \text{then} \quad F^+ = \begin{Bmatrix} A \to B \\ A \to C \\ A \to K \\ B \to K \end{Bmatrix}$$

The closure of F can be systematically determined using the inference rules;

(Note: $F \models X \to Y$ denotes that $X \to Y$ can be inferred from F

(1) reflexive rule: If $X \supseteq Y$, then $X \to Y$

(2) augmentation rule

a) $\{X \to Y\} \models XZ \to YZ$

b) $\{X \to Y\} \models XZ \to Y$

(3) transitive rule: $\{X \to Y, Y \to Z\} \models X \to Z$

(4) decomposition rule: $\{X \to YZ\} \models X \to Y$

(5) additive rule: $\{X \to Y, X \to Z\} \models X \to YZ$

(6) pseudotransitive rule: $\{X \to Y, WY \to Z\} \models WX \to Z$

When decomposing a schema we must make sure there are no lossless joins. This can be done as follows:

Let $R \triangleq$ schema we are decompose
$R_1 + R_2 \triangleq$ two decomposed schemes. (Note $R_1 \cup R_2 = R$)

The decomposition is a lossless-join decomposition if one of the two is true:

a) $R_1 \cap R_2 \rightarrow R_1$

b) $R_1 \cap R_2 \rightarrow R_2$

Note Ant-schema $\cap$ Loan-schema $=$ amount

amount $\not\rightarrow$ Ant-schema

amount $\not\rightarrow$ Loan-schema

hence, decomposition may lose information & is rejected.

Dependency Preservation

The goal is to preserve functional dependencies without requiring a join. (Important if DBMS checks validity of dependencies after inserts, deletes, modifications)

- First, let F = { of all function dependencies in R }
- The closure of F, denoted F^+, is the set of all logically implied dependencies. ~~(...)~~

Ex: $F = \begin{Bmatrix} A \rightarrow B \\ A \rightarrow C \\ B \rightarrow H \end{Bmatrix}$, then $F^+ = \begin{Bmatrix} A \rightarrow B \\ A \rightarrow C \\ A \rightarrow H \\ B \rightarrow H \end{Bmatrix}$

1) reflexivity: $\beta \subset \alpha$, then $\alpha \rightarrow \beta$
2) augmentation: if $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$
3) transitivity: if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$

(1), (2), + (3) are known as Armstrong's axioms.

— Armstrong Axioms are complete, i.e. by repeatedly applying (1)→(3) F^+ can be determined.

Algorithm for determining closure of attribute set α

```
result = α
while (changes to result)
{
    for each FD β → γ in F
        if β ⊆ result
            then result = result ∪ γ
}
```

Example:

assume $F = $
$$A \to B$$
$$A \to C$$
$$CG \to H$$
$$CG \to I$$
$$B \to H$$

A^+ (the closure of A under F) can be calculat

	result	
	A	$A \to B$
iter 1	A, B	$A \to C$
iter 1	A, B, C	$CG \to H$ no, $CG \to I$ no, $B \to H$ yes
iter 1	A B C H	

thus $A^+ = \{A, B, C, H\}$

How about $\{AG\}^+$?

iter 1 : A, B, C, H, I

iter 2 no change

Note $F^+ = \{A, B, C, G, H, I\}^+$

* In worst case this algorithm may take time quadratic in the size of F. There is a linear (but more complex) algorithm

- Often FD are used to make sure the DB does not move into an inconsistent state by the insertion of a tuple or update

- In order to minimize the number of FD that need to be tested, restrict F into a smaller (but just as meaningful) set.
  (1) combine several FD into one where possible
  (2) remove extraneous attributes

$\alpha \to \beta$

- Attribute A is extraneous in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \to \beta\}) \cup \{(\alpha - A) \to \beta\}$

- Attribute A is extraneous in β $(\alpha \to \beta)$ if $A \in \beta$ and F logically implies $(F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\}$

- A <u>canonical cover</u> (or minimal cover) F_C is a set of FD such that F logically implies all FD in F_C, and F_C logically implies all FD in F. Futhermore, F_C must have the following properties:
  (i) no FD in F_C contains an extraneous attribute
  (ii) every FD has a single value on the R.H.S.
  (iii) we can not remove a FD and still be equivalent to F

Example: calculate F_c for F =

$A \to BC$
$B \to C$
$A \to B$
$AB \to C$

1) Make R.H.S have single values only

$$\hat{F} = \begin{matrix} A \to B \\ A \to C \\ B \to C \\ \cancel{A \to B} \\ AB \to C \end{matrix}$$

2) A is extraneous in $AB \to C$ since $B \to C$ logically implies $AB \to C$, thus $(\hat{F} - \{AB \to C\}) \cup \{B \to C\}$ logically implies F^+

removing A from $AB \to C$ gives $B \to C$ which is already in result

$$\hat{F} \Rightarrow \begin{cases} A \to B \\ A \to C \\ B \to C \end{cases}$$

3) We can remove $A \to C$ and still be equivalent to F

$$\hat{F} - \{A \to C\} \Rightarrow F^+$$

$$\Rightarrow F_c = \begin{cases} A \to B \\ B \to C \end{cases}$$

Minimal Cover

19.8e

Formal Algorithm: (note, in this algorith $A \to BCD$ is replaced with $A \to B, A \to C, A \to D$

1) Set $G = F$
2) replace each FD $X \to A_1, A_2, \ldots A_n$ in G by FDs $X \to A_1$, $X \to A_2$, $\ldots$ $X \to A_n$
3) (4) For each FD $X \to A$ in G
   - compute X^+ with respect to $\{G - (X \to A)\}$
   - if X^+ contains A, remove $X \to A$ from G
4) (3) for each remaining FD $X \to A \in G$
   - for each attribute B ~~that~~ is an element of X
     - ~~compute $(X-B)^+$ with respect to $(G - (X \to A)) \cup ((X-B) \to A)$~~
     - ~~if $(X-B)^+$ contains A, replace $X \to A$ with $(X-B) \to A$ in G~~
     - remove B if extraneous

These methods can be used to determine if a relation schema decomposition is dependency preserving.

Minimal Cover Example

19.8f

$$F = \left\{ \begin{array}{ll} ABH \to C & F \to AD \\ A \to D & E \to F \\ C \to E & BH \to E \\ BGH \to F & \end{array} \right\}$$

(a) $\Downarrow$ singleton **STEP 1**

1. $ABH \to C$
2. $A \to D$
3. $C \to E$
4. $\cancel{BGH \to F}$
5. $F \to A$
6. $\cancel{F \to D}$
7. $E \to F$
8. $BH \to E$

STEP 2

See if can remove FDs

1) - Does $(ABH)^+ \ni C$ if (1) is removed?

<u>No</u> → can not remove (1)

2) - Does $A^+ \ni D$ if remove (2) → No

3) - Does $C^+ \ni E$ if remove (3) → No

4) - Does $(BGH)^+ \ni F$ if remove (4)?

<u>Yes</u>, <u>Can</u> remove (4)

5) - $F^+ \ni A$ if remove (5)? No

6) - $F^+ \ni D$ " " (6)? <u>yes</u> can remove 6

7) - No

8) - No

1988

STEP 2 Remove extraneous ATTR

(1) Is A extraneous in $ABH \to C$?

~~If [illegible] replace $ABH \to C$ by $BH \to C$~~

Yes, if $BH \to C$ can be derived from ~~F - (ABH → C) ∪ (BH → C)~~ F'

$BH \to E$

$E \to F \Rightarrow BH \to A \Rightarrow$ ~~ABH~~ $\to A$

$F \to AD$

$\Downarrow$ ✓

$ABH \to C$

(1) $\Rightarrow$ $BH \to C$

Is B extraneous in $BH \to C$? NO

Is H extraneous in $BH \to C$? No

(3) (1) $\Rightarrow$ $BH \to C$

(4) Is G extraneous in $BGH \to F$?

19.8 b

$$\left.\begin{array}{l} BH \to E \\ E \to F \end{array}\right\} BH \to F$$

thus $BGH \to F$, the G is extra

$\Rightarrow BH \to F$

End of step 2

$BH \to C$

$A \to D$

$C \to E$

~~$BH \to F$~~ $\Rightarrow$ step 3 remove redundant FDs

$F \to A$

~~$F \to D$~~

$E \to F$

~~$BH \to E$~~

minimal cover

$BH \to C$

$A \to D$

$C \to E$

$F \to A$

$E \to F$

Now, how about schema decomposition:

Let R be decomposed into $R_1, R_2, \ldots R_n$

The restriction of F to R_i is the set F_i of all functional dependencies that include only attributes of R_i

Let $F' = F_1 \cup F_2 \ldots \cup F_n$

If $F'^+ = F^+$ the decomposition is dependency preserving

Example of how this is used:

Banker-scheme = (branch-name, customer-name, banker-name)

$$F = \begin{cases} \text{banker} \rightarrow \text{branch} \\ \text{customer, branch} \rightarrow \text{banker} \end{cases}$$

Note, key = (customer, branch)

Banker-scheme is not BCNF. Why?

banker → branch, banker is not a super key + the dependency is non-trivial.

⇓ decompose

Banker-Branch-Scheme(banker, branch)
Customer-Banker-Scheme(customer, banker)

This is BCNF + lossless, but does it preserve dependency?

NO!

$$F_1 = \{ \text{banker} \rightarrow \text{branch} \}$$

$$F_2 = \emptyset \quad \text{(only trivial dependencies)}$$

$$\Rightarrow \quad F'^{+} \neq F^{+}$$

Thus, we could not satisfy

a) BCNF, and

b) lossless, and

c) dependency preserving

But, we can get 3NF, lossless, dependency-preserving since

Banker-Scheme <u>is</u> 3NF

(<u>customer, branch</u>, banker)

① $X \rightarrow B$, X is a superkey

② $B \rightarrow A$, A is a subset of a candidate key

Advantage of Normal Form approach:

→ It is a way to formalize a process that is usually only done using intuition

(1) Try to decompose R into BCNF (lossless)

Algorithm:

1) set $D = \{R\}$
2) while $\exists$ schema $Q \in D$ that is not BCNF
   {
   a) choose $Q \in D$ that is not BCNF
   b) find FD $X \to Y$ that violates BCNF
   c) replace Q by two schemas $(Q - Y)$ and $(X \cup Y)$
   }

Note, the final decomposition may not be ~~BCNF~~ dependency preserving

(2) ⇒ test to see if dependency preserving

(3) If not dependency preserving, decompose into lossless & dependency-preserving 3NF

Algorithm

1) Find minimal cover F_c for F
2) for each left hand side $X \in F_c$
   create a relation schema $(X \cup A_1 \cup A_2 \cup \dots A_n)$
   when $X \to A_1, X \to A_2, \dots X \to A_n$ are all the dependencies $\in F_c$ with X in left hand side
3) replace all remaining (unplaced) attributes into a single relation
4) If none of the relation schemas contains a key of R, create one more relation schema that contains attributes that form a key

Example:

Lots (property-id, county-name, lot#, area, price, tax-rate)

$\Downarrow$ abbreviate

Lots (pid, cname, lot#, area, price, tax)

$$F \begin{cases} pid \rightarrow cname, lot\#, area, price, tax \\ cname, lot\# \rightarrow pid, area, price, tax \\ cname \rightarrow tax \\ area \rightarrow price \end{cases}$$

} two candidate keys!

Note $F^+ = F$

Is lots BCNF?

No: $\underbrace{cname \rightarrow tax}$ is a partial dependency, not even 3NF!

$\Downarrow$ Decompose (BCNF decompose algorithm)

lots1 (<u>pid</u>, cname, lot#, area, price) lots2 (<u>cname</u>, tax)

Are both of these BCNF?

No: $\underbrace{area \rightarrow price}$ is part of a transitive dependency!

$\Downarrow$

lots1a (<u>pid</u>, cname, lot#, area) lots1b (<u>area</u>, price) lots2 (<u>cname</u>, tax)

BCNF? Yes!

But is it Dependency Preserving?

$$F_1^{(lots1a)} = \left\{ \begin{array}{l} pid \rightarrow cname,\ lot\#,\ area \\ cname,\ lot\# \rightarrow pid,\ area \end{array} \right.$$

$$F_2^{(lots1b)} = \left\{ area \rightarrow price \right.$$

$$F_3 = \left\{ cname \rightarrow tax \right.$$

$$F'^{+} = \left\{ \begin{array}{l} pid \rightarrow cname,\ lot\#,\ area \\ cname,\ lot\# \rightarrow pid,\ area \\ area \rightarrow price \\ cname \rightarrow tax \end{array} \right.^{+}$$

$$= F^{+}$$

So Yes! it is dependency preserving

Lets add a 5th FD to our example:

area → cname

In this case lots1a is <u>not</u> BCNF

⇓ Decompose

① lots1ax (<u>pid</u>, area, lot#) ② lots1ay (<u>area</u>, cname)

③ lots1b (<u>area</u>, price) ④ lots2 (<u>cname</u>, tax)

Is it BCNF? Yes

Is it dependency preserving?

F_1 = pid → area, lot#
F_2 = area → cname
F_3 = area → price
F_4 = cname → tax

F'^+ = { pid → cname, lot#, area, price, tax
cname → tax
area → price, cname, tax

but <u>NOT</u> cname, lot# → pid, area, price, tax !

<u>Not Dependency Preserving</u>

- Thus, we must settle for 3NF, dependency preserving, lossless and hence use fast decomposition algorithm:

$$F = \begin{cases} pid \to cname, lot\#, area, price, tax \\ cname, lot\# \to pid, area, price, tax \\ cname \to tax \\ area \to price \\ area \to cname \end{cases}$$

Find minimal cover:

1 ~~pid → cname~~
2 " → lot#
3 " → area
4 ~~" → price~~
5 ~~" → tax~~
6 cname, lot# → pid
7 ~~" → area~~
8 ~~" → price~~
9 ~~" → tax~~
10 area → price
11 area → cname
12 cname → tax

pid^+ (with respect to $G - \{pid \to cname\}$)

= l#, area, cname, price, tax

this includes cname, hence can remove 1

- can not remove 2
not 3
yes 4
yes 5
not 6
yes 7
yes 8
yes 9
not 10
not 11
not 12

⟹

pid → l#
" → area
cname, l# → pid
area → price
area → cname
cname → tax

$$F^c \equiv \text{minimal cover} = \left\{ \begin{array}{l} pid \to l\# \\ pid \to area \\ (cname, l\#) \to pid \\ area \to price \\ area \to cname \\ cname \to tax \end{array} \right.$$

3NF, lossless, preserving dep. cont'd:

① (<u>pid</u>, l#, area) ② (<u>area</u>, price, cname)

③ (<u>cname, l#</u>, pid) ④ (<u>cname</u> → tax)

It is 3NF!

(note, not BCNF because <u>cname, l#</u>, pid)

19.17

① Check that it is dependency preserving:

$F_1 =$ pid → l#, area

$F_2 =$ area → price, cname

$F_3 =$ cname, l#, → pid

$F_4 =$ cname → tax

$F'^{+} = \begin{cases} \text{pid} \to \text{l\#, area, price, cname, tax} \\ \text{cname, }\cancel{\text{l\#}} \to \text{pid, area, price, tax} \\ \text{cname} \to \text{tax} \\ \text{area} \to \text{price, cname, tax} \end{cases}$

✓