Relational Design Theory

Relational Design Theory

As noted earlier, the relational model is:

- simple, uniform, well-defined, formal, ...

Such properties tend to lead to useful mathematical theories.

One important theory developed for the relational model involves the notion of functional dependency (*fd*).

Like constraints, assertions, etc. functional dependencies are drawn from the semantics of the application domain.

Essentially, *fd* 's describe how individual attributes are related.

Relational Design Theory (cont)

Functional dependencies

- are a kind of constraint among attributes within a relation
- that have implications for "good" relational schema design

What we study here:

- basic theory and definition of functional dependencies
- methodology for improving schema designs (normalisation)

The aim of studying this:

- improve understanding of relationships among data
- gain enough formalism to assist practical database design

Relational Design and Redundancy

A good relational database design:

- must capture *all* of the necessary attributes/associations
- should do this with a *minimal* amount of stored information

Minimal stored information ⇒ no redundant data.

In database design, redundancy is generally a "bad thing":

- causes problems maintaining consistency after updates

However, it can sometimes lead to performance improvements

- e.g. may be able to avoid a join to collect bits of data together

---

Relational Design and Redundancy (cont)

Consider the following relation defining bank accounts/branches:

accountNo	balance	customer	branch	address	assets
A–101	500	1313131	Downtown	Brooklyn	9000000
A–102	400	1313131	Perryridge	Horseneck	1700000
A–113	600	9876543	Round Hill	Horseneck	8000000
A–201	900	9876543	Brighton	Brooklyn	7100000
A–215	700	1111111	Mianus	Horseneck	400000
A–222	700	1111111	Redwood	Palo Alto	2100000
A–305	350	1234567	Round Hill	Horseneck	8000000

We need to be careful updating this data, otherwise we may introduce inconsistencies.

Relational Design and Redundancy (cont)

Insertion anomaly:

- when we insert a new record, we need to check that branch data is consistent with existing tuples

Update anomaly:

- if a branch changes address, we need to update all tuples referring to that branch

Deletion anomaly:

- if we remove information about the last account at a branch, all of the branch information disappears

(If we do manage to avoid inconsistencies, the cost is additional updates)

Relational Design and Redundancy (cont)

Insertion anomaly example (insert account A–306 at Round Hill):

accountNo	balance	customer	branch	address	assets
A–101	500	1313131	Downtown	Brooklyn	9000000
A–102	400	1313131	Perryridge	Horseneck	1700000
A–113	600	9876543	Round Hill	Horseneck	8000000
A–201	900	9876543	Brighton	Brooklyn	7100000
A–215	700	1111111	Mianus	Horseneck	400000
A–222	700	1111111	Redwood	Palo Alto	2100000
A–305	350	1234567	Round Hill	Horseneck	8000000
A–306	800	1111111	Round Hill	Horseneck	8000800

Relational Design and Redundancy (cont)

Update anomaly example (update Round Hill branch address):

accountNo	balance	customer	branch	address	assets
A–101	500	1313131	Downtown	Brooklyn	9000000
A–102	400	1313131	Perryridge	Horseneck	1700000
A–113	600	9876543	Round Hill	Palo Alto	8000000
A–201	900	9876543	Brighton	Brooklyn	7100000
A–215	700	1111111	Mianus	Horseneck	400000
A–222	700	1111111	Redwood	Palo Alto	2100000
A–305	350	1234567	Round Hill	Horseneck	8000000

Relational Design and Redundancy (cont)

Deletion anomaly example (remove account A–101):

accountNo	balance	customer	branch	address	assets
A–101	500	1313131	Downtown	Brooklyn	9000000
A–102	400	1313131	Perryridge	Horseneck	1700000

A-113	600	9876543	Round Hill	Horseneck	8000000
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hill	Horseneck	8000000

Where is the Downtown branch located? What are its assets?

Database Design (revisited)

To avoid these kinds of update problems:

- decompose the relation U into several smaller relations R_i
- where each R_i has minimal overlap with other R_j

Typically, each R_i contains information about one entity (e.g. branch, customer, ...)

This leads to a (bottom-up) database design procedure:

- start from an unstructured collection of attributes
- use normalisation (via *fd*s) to impose structure
- final schema is a collection of tables
- final schema has minimal redundancy (normalised)

Database Design (revisited) (cont)

This contrasts with our earlier (top-down) design procedure:

- structure data at conceptual level (ER design)
- then map to "physical" level (relational design)
- final schema is a collection of tables

It appears that ER-design-then-relational-mapping

- leads to a collection of well-structured tables
- which is similar to a normalised schema

So why do we need a dependency theory and normalisation procedure to deal with redundancy?

Database Design (revisited) (cont)

Some reasons ...

1. ER design does not guarantee minimal redundancy
   - dependency theory allows us to check designs for residual problems
2. Normalisation can be viewed as (semi)automated design
   - determine all of the attributes in the problem domain
   - collect them all together in a "super-relation" (with update anomalies)
   - provide information about how attributes are related
   - apply normalisation to decompose into non-redundant relations

---

Notation/Terminology

Most texts adopt the following terminology:

Relation schemas	upper-case letters, denoting set of all attributes (e.g. *R*, *S*, *P*, *Q*)
Relation instances	lower-case letter corresponding to schema (e.g. *r(R)*, *s(S)*, *p(P)*, *q(Q)*)
Tuples	lower-case letters (e.g. t, t', t_1, u, v)
Attributes	upper-case letters from start of alphabet (e.g. *A*, *B*, *C*, *D*)
Sets of attributes	simple concatenation of attribute names (e.g. *X=ABCD*, *Y=EFG*)
Attributes in tuples	tuple[attrSet] (e.g. *t[ABCD]*, *t[X]*)

Functional Dependency

A relation instance *r(R)* satisfies a dependency $X \rightarrow Y$ if

- for any $t, u \in r$, $\quad t[X] = u[X] \quad \Rightarrow \quad t[Y] = u[Y]$

In other words, if two tuples in *R* agree in their values for the set of attributes *X*, then they must also agree in their values for the set of attributes *Y*.

We say that "*Y* is functionally dependent on *X*".

Attribute sets *X* and *Y* may overlap; trivially true that $X \rightarrow X$.

Notes:

- the single arrow $\rightarrow$ denotes "functional dependency"
- $X \rightarrow Y$ can also be read as "*X* determines *Y*"
- the double arrow $\Rightarrow$ denotes "logical implication"

Functional Dependency (cont)

The above definition talks about dependency within a relation instance *r(R)*.

Much more important for design is the notion of dependency across all possible instances of the relation (i.e. a schema-based dependency).

This is a simple generalisation of the previous definition:

- for any $t, u \in$ ***any*** $r(R)$, $t[X] = u[X] \Rightarrow t[Y] = u[Y]$

Useful because such dependencies reflect the semantics of the problem.

Functional Dependency (cont)

Consider the following instance *r(R)* of the relation schema *R(ABCDE)*:

A	*B*	*C*	*D*	*E*
a_1	b_1	c_1	d_1	e_1
a_2	b_1	c_2	d_2	e_1
a_3	b_2	c_1	d_1	e_1
a_4	b_2	c_2	d_2	e_1

a_5	b_3	c_3	d_1	e_1

What kind of dependencies can we observe among the attributes in r(R)?

Functional Dependency (cont)

Since the values of *A* are unique, it follows from the *fd* definition that:

$A \to B, \quad A \to C, \quad A \to D, \quad A \to E$

It also follows that $A \to BC$ (or any other subset of *ABCDE*).

This can be summarised as $A \to BCDE$

From our understanding of primary keys, *A* is a PK.

Functional Dependency (cont)

Since the values of *E* are always the same, it follows that:

$A \rightarrow E, \quad B \rightarrow E, \quad C \rightarrow E, \quad D \rightarrow E$

Note: **cannot** generally summarise above by $ABCD \rightarrow E$

(However, $ABCD \rightarrow E$ does happen to be true in this example)

In general, $A \rightarrow Y, \quad B \rightarrow Y \qquad AB \rightarrow Y$

Functional Dependency (cont)

Other observations:

- combinations of BC are unique, therefore $BC \rightarrow ADE$
- combinations of BD are unique, therefore $BD \rightarrow ACE$
- if C values match, so do D values, therefore $C \rightarrow D$
- however, D values don't determine C values, so $D \quad C$

We could derive many other dependencies, e.g. $AE \rightarrow BC, \ldots$

In practice, choose a minimal set of *fd*s (basis)

- from which all other *fd*s can be derived
- which typically captures useful problem-domain information

Functional Dependency (cont)

Can we generalise some ideas about functional dependency?

E.g. are there dependencies that hold for *any* relation?

Yes, but they're rather uninteresting ones such as:

$t[ABC] = u[ABC] \;\Rightarrow\; t[AB] = u[AB]$ giving $ABC \rightarrow AB$

which generalises to $Y \subset X \;\Rightarrow\; X \rightarrow Y$.

E.g. do some dependencies suggest the existence of others?

Yes, and this is much more interesting ... there are a number of rules of inference that allow us to derive dependencies.

Inference Rules

Armstrong's rules are complete, general rules of inference on *fd*s.

F1. Reflexivity e.g. $X \rightarrow X$

- a formal statement of *trivial dependencies*; useful for derivations

F2. Augmentation e.g. $X \rightarrow Y \;\Rightarrow\; XZ \rightarrow YZ$

- if a dependency holds, then we can freely expand its left hand side

F3. Transitivity e.g. $X \rightarrow Y,\ Y \rightarrow Z \;\Rightarrow\; X \rightarrow Z$

- the "most powerful" inference rule; useful in multi-step derivations

---

Inference Rules (cont)

While Armstrong's rules are complete, other useful rules exist:

F4. Additivity e.g. $X \rightarrow Y, X \rightarrow Z \;\Rightarrow\; X \rightarrow YZ$

- useful for constructing new right hand sides of *fd*s (also called union)

F5. Projectivity e.g. $X \to YZ \;\Rightarrow\; X \to Y,\, X \to Z$

- useful for reducing right hand sides of *fd*s (also called decomposition)

F6. Pseudotransitivity e.g. $X \to Y,\, YZ \to W \;\Rightarrow\; XZ \to W$

- shorthand for a common transitivity derivation

Inference Rules (cont)

Using rules and a set F of given *fd*s, we can determine what other *fd*s hold.

Example (derivation of $AB \to GH$):

$R = ABCDEFGHIJ$

$F = \{\, AB \to E,\quad AG \to J,\quad BE \to I,\quad E \to G,\quad GI \to H \,\}$

1. $AB \to E$ (given)

2. $AB \to AB$ (using F1)
3. $AB \to B$ (using F5 on 2)
4. $AB \to BE$ (using F4 on 1,3)
5. $BE \to I$ (given)

Inference Rules (cont)

Continuing the derivation ...

6. $AB \to I$ (using F3 on 4,5)
7. $E \to G$ (given)
8. $AB \to G$ (using F3 on 1,7)
9. $AB \to GI$ (using F4 on 6,8)
10. $GI \to H$ (given)
11. $AB \to H$ (using F3 on 6,8)
12. $GI \to GI$ (using F1)

13. $GI \rightarrow I$ (using F5 on 12)
14. $AB \rightarrow GH$ (using F4 on 8,11)

Closures

Given a set F of *fd*s, how many new *fd*s can we derive?

For a finite set of attributes, there must be a finite set of *fd*s.

The largest collection of dependencies that can be derived from F is called the closure of F and is denoted F^+.

Closures allow us to answer two interesting questions:

- is a particular dependency $X \rightarrow Y$ derivable from F?
- are two sets of dependencies F and G equivalent?

Closures (cont)

For the question "is $X \rightarrow Y$ derivable from F?" ...

- compute the closure F^+; check whether $X \rightarrow Y \quad \in \quad F^+$

For the question "are F and G equivalent?" ...

- compute closures F^+ and G^+; check whether they're equal

Unfortunately, closures on even small sets of functional dependencies can be very large.

Algorithms based on F^+ rapidly become infeasible.

Closures (cont)

Example (of *fd* closure):

$R = ABC, \quad F = \{ AB \rightarrow C, \quad C \rightarrow B \}$

$F^+ = \{ A \rightarrow A, \quad AB \rightarrow A, \quad AC \rightarrow A, \quad AB \rightarrow B, \quad BC \rightarrow B, \quad ABC \rightarrow B,$
$\qquad C \rightarrow C, \quad AC \rightarrow C, \quad BC \rightarrow C, \quad ABC \rightarrow C, \quad AB \rightarrow AB, \quad . \ . \ . \ . \ . \ . \ . \ ,$
$\qquad AB \rightarrow ABC, \quad AB \rightarrow ABC, \quad C \rightarrow B, \quad C \rightarrow BC, \quad AC \rightarrow B, \quad AC \rightarrow AB \}$

To solve this problem, use closures based on sets of attributes rather than sets of *fd*s.

Given a set X of attributes and a set F of *fd*s, the largest set of attributes that can be derived from X using F, is called the closure of X (denoted X^+).

We can prove (using additivity) that $(X \rightarrow Y) \in F^+$ iff $Y \subset X^+$.

For computation, $|X^+|$ is bounded by the number of attributes.

Closures (cont)

For the question "is $X \rightarrow Y$ derivable from F?" ...

- compute the closure X^+, check whether $Y \subset X^+$

For the question "are F and G equivalent?" ...

- for each dependency in G, check whether derivable from F

- for each dependency in F, check whether derivable from G
- if true for all, then $F \Rightarrow G$ and $G \Rightarrow F$ which implies $F^+ = G^+$

Closure Algorithm

```
Inputs: set F of fds
        set X of attributes
Output: closure of X (i.e. X⁺)

X⁺ = X
stillChanging = true;
while (stillChanging) {
    stillChanging = false;
    for each W → Z in F {
        if (W ⊆ X⁺) and not (Z ⊆ X⁺) {
            X⁺ = X⁺ ∪ Z
            stillChanging = true;
        }
    }
}
```

Closure Algorithm (cont)

E.g. R = $ABCDEF$, $F = \{ AB \rightarrow C,\ \ BC \rightarrow AD,\ \ D \rightarrow E,\ \ CF \rightarrow B \}$

Does $AB \rightarrow D$ follow from F? Solve by checking $D \in AB^+$.

Computing AB^+:

1. $AB^+ = AB$ (initially)
2. $AB^+ = ABC$ (using $AB \rightarrow C$)
3. $AB^+ = ABCD$ (using $BC \rightarrow AD$)
4. $AB^+ = ABCDE$ (using $D \rightarrow E$)

Since D is in AB^+, then $AB \rightarrow D$ does follow from F.

Closure Algorithm (cont)

E.g. $R = ABCDEF$, $F = \{ AB \to C,\ \ BC \to AD,\ \ D \to E,\ \ CF \to B \}$

Does $D \to A$ follow from F? Solve by checking $A \in D^+$.

Computing D^+:

1. $D^+ = D$ (initially)
2. $D^+ = DE$ (using $D \to E$)

Since A is not in D^+, then $D \to A$ does not follow from F.

Closure Algorithm (cont)

E.g. $R = ABCDEF$, $F = \{ AB \to C,\ \ BC \to AD,\ \ D \to E,\ \ CF \to B \}$

What are the keys of R?

Solve by finding $X \subset R$ such that $X^+ = R$.

From previous examples, we know AB and D are not keys.

This also implies that A and B alone are not keys.

So how to find keys? Try all combinations of $ABCDEF$ ...

E.g. maybe ACF is a key ...

Closure Algorithm (cont)

Computing ACF^+:

1. $ACF^+ = ACF$ (initially)
2. $ACF^+ = ABCF$ (using $CF \rightarrow B$)
3. $ACF^+ = ABCDF$ (using $BC \rightarrow AD$)
4. $ACF^+ = ABCDEF$ (using $D \rightarrow E$)

Since $ACF^+ = R$, ACF is a key (as is ABF).

Minimal Covers

For a given application, we can define many different sets of *fd*s with the same closure (e.g. F and G where $F^+ = G^+$)

Which one is best to "model" the application?

- any model has to be complete (i.e. capture entire semantics)
- models should be as small as possible
  (we use them to check DB validity after update; less checking is better)

If we can ...

- determine a number of candidate *fd* sets, F, G and H
- establish that $F^+ = G^+ = H^+$
- we would then choose the smallest one for our "model"

Better still, can we *derive* the smallest complete set of *fd*s?

Minimal Covers (cont)

Minimal cover F_c for a set F of *fd*s:

- F_c is equivalent to F
- all *fd*s have the form $X \rightarrow A$ (where A is a single attribute)
- it is not possible to make F_c smaller
  - either by deleting an *fd*
  - or by deleting an attribute from an *fd*

An *fd* d is redundant if $(F - \{d\})^+ = F^+$

An attribute a is redundant if $(F - \{d\} \cup \{d'\})^+ = F^+$
(where d' is the same as d but with attribute A removed)

Minimal Covers (cont)

Algorithm for computing minimal cover:

```
Inputs: set F of fds
Output: minimal cover Fc of F

Fc = F
Step 1:
    put f ∈ Fc into canonical form
Step 2:
    eliminate redundant attributes from f ∈ Fc
Step 3:
    eliminate redundant fds from Fc
```

Minimal Covers (cont)

Step 1: put *fd*s into canonical form

```
for each f ∈ Fc like X → {A1,..,An}
    Fc = Fc - {f}
    for each a in {A1,..,An}
        Fc = Fc ∪ {X → a}
    end
end
```

Minimal Covers (cont)

Step 2: eliminate redundant attributes

```
for each f ∈ Fc like X → A
    for each b in X
        f' = (X-{b}) → A;
        G = Fc - {f} ∪ {f'}
        if (G+ == Fc+) Fc = G
    end
end
```

Minimal Covers (cont)

Step 3: eliminate redundant functional dependencies

```
for each f ∈ Fc
    G = Fc - {f}
    if (G+ == Fc+) Fc = G
end
```

Note: we often assume that any supplied F is minimal.

Minimal Covers (cont)

E.g. $R = ABC$, $F = \{ A \rightarrow BC,\ \ B \rightarrow C,\ \ A \rightarrow B,\ \ AB \rightarrow C \}$

Compute the minimal cover:

- canonical *fd*s: $A \rightarrow B,\ \ A \rightarrow C,\ \ B \rightarrow C,\ \ AB \rightarrow C$
- redundant attrs: $A \rightarrow B,\ \ A \rightarrow C,\ \ B \rightarrow C,\ \ AB \rightarrow C$
- redundant *fd*s: $A \rightarrow B,\ \ A \rightarrow C,\ \ B \rightarrow C$

This gives the minimal cover $F_c = \{ A \rightarrow B,\ \ B \rightarrow C \}$.

Normalization

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.

Normal Forms

Normalization theory defines six normal forms (NFs).

Each normal form:

- involves a set of dependency properties that a schema must satisfy
- gives guarantees about presence/absence of update anomalies

Higher normal forms have less redundancy ⇒ less update problems.

Normal Forms (cont)

Must first decide which normal form *rNF* is "acceptable".

The normalization process:

- check whether each relation in schema is in *rNF*
- if a relation is not in *rNF*
  - partition into sub-relations where each is closer to *rNF*
- repeat until all relations in schema are in *rNF*

Normal Forms (cont)

A brief history of 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 hierarachy: 5NF ⇒ 4NF ⇒ BCNF ⇒ 3NF ⇒ 2NF ⇒ 1NF

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

Normal Forms (cont)

1NF	all attributes have atomic values we assume this as part of relational model
2NF	all non-key attributes fully depend on key (i.e. no partial dependencies) avoids much redundancy
3NF BCNF	no attributes dependent on non-key attrs (i.e. no transitive dependencies) avoids remaining redundancy
4NF	removes problems due to multivalued dependencies
5NF	removes problems due to join dependencies

Normal Forms (cont)

In practice, BCNF and 3NF are the most important.
(these are generally the "acceptable normal forms" for relational design)

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

Relation Decomposition

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 $r(R) = s(S)$ $t(T)$

We may require several decompositions to achieve acceptable NF.

Normalization algorithms tell us how to choose S and T.

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 (cont)

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

The cause of these problems can be stated in terms of *fd*s

- a branch is located in one city   *branchName* → *branchCity*
- a branch may handle many loans   *branchName 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 (cont)

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 (cont)

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

Schema Design (cont)

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

Schema Design (cont)

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.

Schema Design (cont)

The result of *Join(BranchCust,CustLoan)*

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 (cont)

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.

Boyce-Codd Normal Form

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

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

Observations:

- any two-attribute relation is in BCNF
- any relation with key K, other attributes X, and $K \rightarrow X$ is in BCNF

Boyce-Codd Normal Form (cont)

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

- no update anomalies due to *fd*-based redundancy
- lossless join decomposition

However, we are not guaranteed:

- 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
}
```

BCNF Decomposition (cont)

Example (the *BankLoans* schema):

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

Has functional dependencies *F*

- *branchName* → *assets,branchCity*
- *loanNo* → *amount,branchName*

The key for *BankLoans* is *branchName,custName,loanNo*

BCNF Decomposition (cont)

Applying the BCNF algorithm:

- check *BankLoans* relation ... it is not in BCNF

  (*branchName* → *assets,branchCity* violates BCNF criteria; LHS is not a key)

- to fix ... decompose *BankLoans* into

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

- check *Branch* relation ... it is in BCNF

  (the only nontrivial *fd*s have LHS=*branchName*, which is a key)

(continued)

BCNF Decomposition (cont)

Applying the BCNF algorithm (cont):

- check *LoanInfo* relation ... it is not in BCNF

  (*loanNo* → *amount,branchName* violates BCNF criteria; LHS is not a key)

- to fix ... decompose *LoanInfo* into

> *Loan(branchName, loanNo, amount)*
> *Borrower(custName, loanNo)*

- check *Loan* ... it is in BCNF
- check *Borrower* ... it is in BCNF

Third Normal Form

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

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 (cont)

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 (cont)

The following algorithm converts an arbitrary schema to 3NF:

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

let Fc be a minimal cover for F
Res = {}
for each fd X → Y in Fc {
    if (no schema S ∈ Res contains XY) {
        Res = Res ∪ XY
    }
}
if (no schema S ∈ Res contains a candidate key for R) {
    K = any candidate key for R
    Res = Res ∪ K
}
```

Database Design Methodology

To achieve a "good" database design:

- identify attributes, entities, relationships → ER design

- map ER design to relational schema
- identify constraints (including keys and functional dependencies)
- apply BCNF/3NF algorithms to produce normalized schema

Note: may subsequently need to "denormalise" if the design yields inadequate performance.

Produced: 13 Sep 2020