Contents

Database design

- Just like a C program, a DB design can be syntactically correct, but low quality
- Properties of good design
  - Easy to understand
  - Safe (not error-prone)
  - Efficient
    - fast queries
    - disk space
  - Good selection of keys, detailed constraints
  - Easy design update and evolution
  - ...

Database design

- Just like a C program, a DB design can be syntactically correct, but low quality
- Properties of good design
  - Easy to understand
  - **Safe** (not error-prone)
  - Efficient
    - fast queries
    - disk space
  - Good selection of keys, detailed constraints
  - Easy design **update** and **evolution**
  - ...
- Formalizable properties: **normal forms**

Goals of this chapter

General

- Define better table designs
- Understand tradeoffs between design quality and efficiency

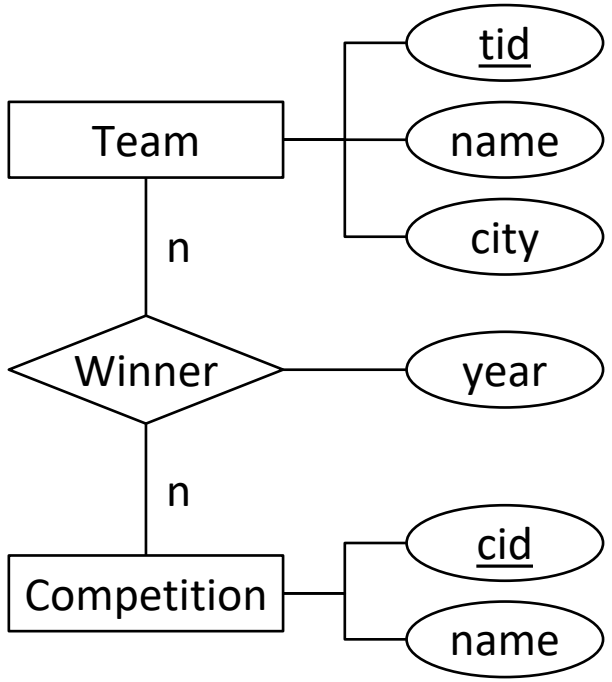
Specific

- Understand the concept of functional dependency
- Understand the definition of normal forms,
  be able to determine the normal form of a table
- Be able to decompose tables into higher normal forms
  systematically using two algorithms

DB design quality: bad design example

TeamWinnerCompetition

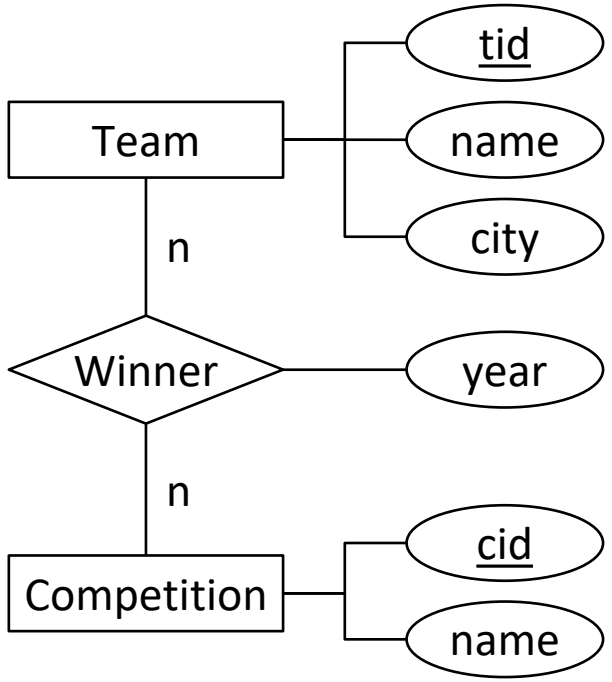
tid	name	city	cid	competition	year
1	Liverpool F.C.	Liverpool	1	English Premier League	2020
1	Liverpool F.C.	Liverpool	1	English Premier League	1990
1	Liverpool	Liverpool	2	FA Cup	2006
2	Manchester City	Manchester	1	English Premier League	2019
2	Manchester City	Manchester	1	English Premier League	2018
2	Manchester City	Manchester	2	FA Cup	2019
2	Manchester City	Manchester	2	FA Cup	2011
3	Rochdale A.F.C	Rochdale	NULL	NULL	NULL
4	Shrewsbury Town F.C.	Shrewsbury	NULL	NULL	NULL



DB design quality: bad design example

TeamWinnerCompetition

tid	name	city	cid	competition	year
1	Liverpool F.C.	Liverpool	1	English Premier League	2020
1	Liverpool F.C.	Liverpool	1	English Premier League	1990
1	Liverpool	Liverpool	2	FA Cup	2006
2	Manchester City	Manchester	1	English Premier League	2019
2	Manchester City	Manchester	1	English Premier League	2018
2	Manchester City	Manchester	2	FA Cup	2019
2	Manchester City	Manchester	2	FA Cup	2011
3	Rochdale A.F.C	Rochdale	NULL	NULL	NULL
4	Shrewsbury Town F.C.	Shrewsbury	NULL	NULL	NULL



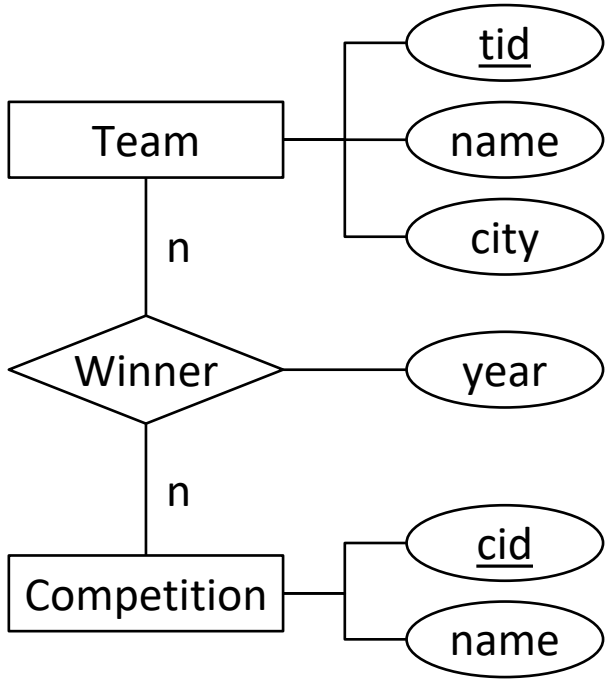
Problems

- **Redundancy**

DB design quality: bad design example

TeamWinnerCompetition

tid	name	city	cid	competition	year
1	Liverpool F.C.	Liverpool	1	English Premier League	2020
1	Liverpool F.C.	Liverpool	1	English Premier League	1990
1	**Liverpool**	Liverpool	2	FA Cup	2006
2	Manchester City	Manchester	1	English Premier League	2019
2	Manchester City	Manchester	1	English Premier League	2018
2	Manchester City	Manchester	2	FA Cup	2019
2	Manchester City	Manchester	2	FA Cup	2011
3	Rochdale A.F.C	Rochdale	NULL	NULL	NULL
4	Shrewsbury Town F.C.	Shrewsbury	NULL	NULL	NULL



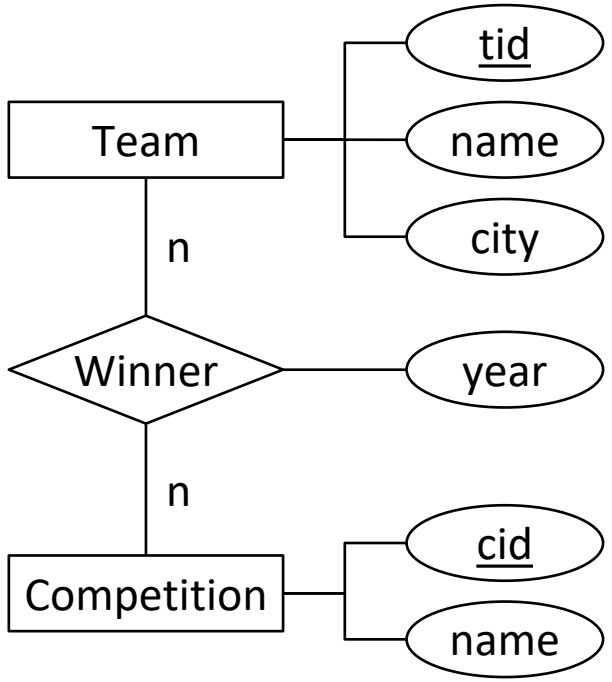
Problems

- Redundancy
- **Error-prone cell value updates**

DB design quality: bad design example

TeamWinnerCompetition

tid	name	city	cid	competition	year
1	Liverpool F.C.	Liverpool	1	English Premier League	2020
1	Liverpool F.C.	Liverpool	1	English Premier League	1990
1	Liverpool F.C.	Liverpool	2	FA Cup	2006
2	Manchester City	Manchester	1	English Premier League	2019
2	Manchester City	Manchester	1	English Premier League	2018
2	Manchester City	Manchester	2	FA Cup	2019
2	Manchester City	Manchester	2	FA Cup	2011
3	Rochdale A.F.C	Rochdale	**NULL**	**NULL**	**NULL**
4	Shrewsbury Town F.C.	Shrewsbury	**NULL**	**NULL**	**NULL**



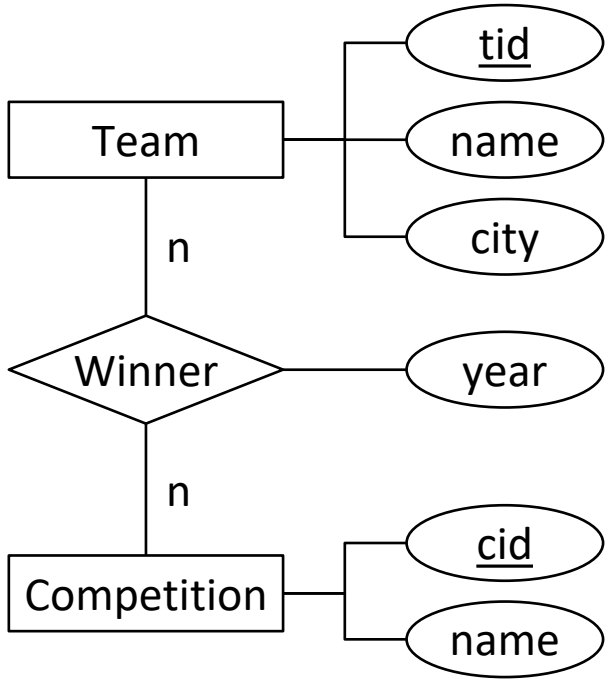
Problems

- Redundancy
- Error-prone cell value updates
- **NULL values**

DB design quality: bad design example

TeamWinnerCompetition

tid	name	city	cid	competition	year
1	Liverpool F.C.	Liverpool	1	English Premier League	2020
1	Liverpool F.C.	Liverpool	1	English Premier League	1990
1	Liverpool F.C.	Liverpool	2	FA Cup	2006
2	Manchester City	Manchester	1	English Premier League	2019
2	Manchester City	Manchester	1	English Premier League	2018
2	Manchester City	Manchester	2	FA Cup	2019
2	Manchester City	Manchester	2	FA Cup	2011
3	Rochdale A.F.C	Rochdale	NULL	NULL	NULL
4	Shrewsbury Town F.C.	Shrewsbury	NULL	NULL	NULL



Problems

- Redundancy
- Error-prone cell value updates
- NULL values

The source of the problems:

tid → { name, city }

cid → competition

where tid or cid alone **are not unique** in the table

What to do

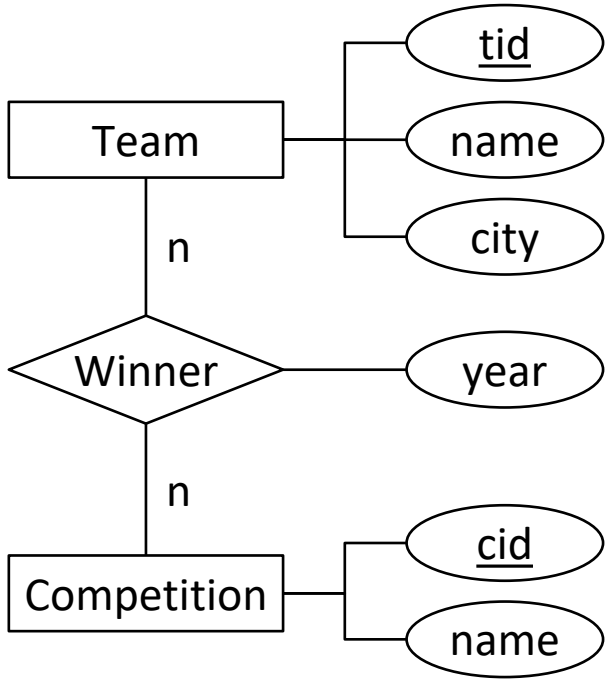
a) Follow the ER conversion to tables procedure, or…

b) Understand functional dependencies, diagnose problems, break down tables

DB design quality: bad design example

TeamWinnerCompetition

tid	name	city	cid	competition	year
1	Liverpool F.C.	Liverpool	1	English Premier League	2020
1	Liverpool F.C.	Liverpool	1	English Premier League	1990
1	Liverpool F.C.	Liverpool	2	FA Cup	2006
2	Manchester City	Manchester	1	English Premier League	2019
2	Manchester City	Manchester	1	English Premier League	2018
2	Manchester City	Manchester	2	FA Cup	2019
2	Manchester City	Manchester	2	FA Cup	2011
3	Rochdale A.F.C	Rochdale	NULL	NULL	NULL
4	Shrewsbury Town F.C.	Shrewsbury	NULL	NULL	NULL



Problems

- Redundancy
- Error-prone cell value updates
- NULL values

The source of the problems:

tid → { name, city }

cid → competition

where tid or cid alone are not unique in the table

What to do

a) Follow the ER conversion to tables procedure, or…

b) Understand **functional dependencies**, diagnose problems, break down tables

Functional dependency

Given a table schema $T(a, b, \dots)$

We say that a column b **functionally depends on** another column a if:

$$t_1.a = t_2.a \Rightarrow t_1.b = t_2.b$$

for any two rows t_1, t_2 in any state of table T

We express this as $\boldsymbol{a \rightarrow b}$, which we call a **functional dependency**
because in a way the value of b is a function of the value of a: $b = f(a)$

Functional dependencies can involve sets of columns: $\{a, b\} \rightarrow \{c, d\}$

Functional dependency examples

- The canonical dependency:

  If X is a **unique** set of columns in a table T then $X \rightarrow a$ for any column a of T

- Examples

User

id	name	email
24	Amelia	amy@gmail.com
6	Amelia	amy2@gmail.com
6	James	jim@gmail.com
81	Nicholas	jim@gmail.com

id $\rightarrow$ { name, email }
email $\rightarrow$ { name, id } $\Longleftarrow$ Unique: id, email

Functional dependency examples

- The canonical dependency:

  If X is a **unique** set of columns in a table T then $X \rightarrow a$ for any column a of T

- Examples

User

id	name	email
24	Amelia	amy@gmail.com
6	Amelia	amy2@gmail.com
6	James	jim@gmail.com
81	Nicholas	jim@gmail.com

Ok

id $\rightarrow$ { name, email }
email $\rightarrow$ { name, id }

$\Longleftarrow$ Unique: id, email

Functional dependency examples

- The canonical dependency:

  If X is a **unique** set of columns in a table T then $X \rightarrow a$ for any column a of T

- Examples

User

	id	name	email
Not ok	24	Amelia	amy@gmail.com
	6	Amelia	amy2@gmail.com
	6	James	jim@gmail.com
	81	Nicholas	jim@gmail.com

id $\rightarrow$ { name, email }
email $\rightarrow$ { name, id } $\Longleftarrow$ Unique: id, email

Functional dependency examples

- The canonical dependency:

  If X is a **unique** set of columns in a table T then $X \rightarrow a$ for any column a of T

- Examples

User

id	name	email
24	Amelia	amy@gmail.com
6	Amelia	amy2@gmail.com
6	James	jim@gmail.com
81	Nicholas	jim@gmail.com

} Not ok

id $\rightarrow$ { name, email }
email $\rightarrow$ { name, id } $\Longleftarrow$ Unique: id, email

Functional dependency examples

- The canonical dependency:

  If X is a **unique** set of columns in a table T then $X \rightarrow a$
  for any column a of T

- Examples

User

id	name	email
24	Amelia	amy@gmail.com
6	Amelia	amy2@gmail.com
6	James	jim@gmail.com
81	Nicholas	jim@gmail.com

Follows

user1	user2	date
6	24	‘2020-10-02’
6	12	‘2020-10-02’
81	12	‘2020-10-02’
12	73	‘2020-10-02’
12	73	‘2020-10-05’

Unique: { user1, user2 } $\Longrightarrow$ { user1, user2 } $\rightarrow$ date

Functional dependency examples

- The canonical dependency:

  If X is a **unique** set of columns in a table T then $X \rightarrow a$
  for any column a of T

- Examples

User

id	name	email
24	Amelia	amy@gmail.com
6	Amelia	amy2@gmail.com
6	James	jim@gmail.com
81	Nicholas	jim@gmail.com

Follows

Ok

user1	user2	date
6	24	‘2020-10-02’
6	12	‘2020-10-02’
81	12	‘2020-10-02’
12	73	‘2020-10-02’
12	73	‘2020-10-05’

Unique: { user1, user2 } $\Longrightarrow$ { user1, user2 } $\rightarrow$ date

Functional dependency examples

- The canonical dependency:

  If X is a **unique** set of columns in a table T then $X \rightarrow a$
  for any column a of T

- Examples

User

id	name	email
24	Amelia	amy@gmail.com
6	Amelia	amy2@gmail.com
6	James	jim@gmail.com
81	Nicholas	jim@gmail.com

Follows

Ok

user1	user2	date
6	24	‘2020-10-02’
6	12	‘2020-10-02’
81	12	‘2020-10-02’
12	73	‘2020-10-02’
12	73	‘2020-10-05’

Unique: { user1, user2 } $\Longrightarrow$ { user1, user2 } $\rightarrow$ date

Functional dependency examples

- The canonical dependency:

  If X is a **unique** set of columns in a table T then $X \rightarrow a$
  for any column a of T

- Examples

User

id	name	email
24	Amelia	amy@gmail.com
6	Amelia	amy2@gmail.com
6	James	jim@gmail.com
81	Nicholas	jim@gmail.com

Follows

user1	user2	date
6	24	‘2020-10-02’
6	12	‘2020-10-02’
81	12	‘2020-10-02’
12	73	‘2020-10-02’
12	73	‘2020-10-05’

Not ok

Unique: { user1, user2 } $\Longrightarrow$ { user1, user2 } $\rightarrow$ date

Functional dependency examples

- The canonical dependency:

  If X is a **unique** set of columns in a table T then $X \rightarrow a$
  for any column a of T

- Examples

User

id	name	email
24	Amelia	amy@gmail.com
6	Amelia	amy2@gmail.com
6	James	jim@gmail.com
81	Nicholas	jim@gmail.com

Follows

user1	user2	date
6	24	‘2020-10-02’
6	12	‘2020-10-02’
81	12	‘2020-10-02’
12	73	‘2020-10-02’
12	73	‘2020-10-05’

Ok

Unique: { user1, user2 } $\Longrightarrow$ { user1, user2 } $\rightarrow$ date

Functional dependency examples

- The canonical dependency:

  If X is a **unique** set of columns in a table T then $X \rightarrow a$
  for any column a of T

- Are there any other possible dependencies?

  In a good table design, no

  In a table design produced following the steps from ER model, no

  In a careless table design, maybe

  More importantly sometimes we trade design quality for query efficiency, and tolerate “improper” dependencies

Functional dependencies in a “bad” design

Follows

user1	name1	user2	name2	date

$\{ \text{user1}, \text{user2} \} \rightarrow \text{date}$

$\text{user1} \rightarrow \text{name1}$

$\text{user2} \rightarrow \text{name2}$

Functional dependencies in a “bad” design

Follows

user1	name1	user2	name2	date
6	James	24	Amelia	‘2020-10-02’
6	Laura	12	David	‘2020-10-02’
81	Nicholas	12		‘2020-10-02’
12	Laura	73		‘2020-10-02’
12		73	Catherine	

$\{ \text{user1}, \text{user2} \} \rightarrow \text{date}$

$\text{user1} \rightarrow \text{name1}$

$\text{user2} \rightarrow \text{name2}$

Functional dependencies in a “bad” design

Follows

user1	name1	user2	name2	date
6	James	24	Amelia	‘2020-10-02’
6	Laura	12	David	‘2020-10-02’
81	Nicholas	12		‘2020-10-02’
12	Laura	73		‘2020-10-02’
12		73	Catherine	

Not ok

{ user1, user2 } → date

user1 → name1

user2 → name2

Functional dependencies in a “bad” design

Follows

user1	name1	user2	name2	date
6	James	24	Amelia	‘2020-10-02’
6		12	David	‘2020-10-02’
81	Nicholas	12		‘2020-10-02’
12	Laura	73		‘2020-10-02’
12		73	Catherine	

$\{ \text{user1}, \text{user2} \} \rightarrow \text{date}$

$\text{user1} \rightarrow \text{name1}$

$\text{user2} \rightarrow \text{name2}$

Functional dependencies in a “bad” design

Follows

user1	name1	user2	name2	date
6	James	24	Amelia	‘2020-10-02’
6		12	David	‘2020-10-02’
81	Nicholas	12		‘2020-10-02’
12	Laura	73		‘2020-10-02’
12		73	Catherine	

{ user1, user2 } → date

user1 → name1

user2 → name2

Not really ok → But doesn’t violate any functional dependency!

Functional dependencies in a “bad” design

Follows

user1	name1	user2	name2	date
6	James	24	Amelia	‘2020-10-02’
6		12	David	‘2020-10-02’
81	Nicholas	12		‘2020-10-02’
12	Laura	73		‘2020-10-02’
12		73	Catherine	

{ user1, user2 } → date

“Good” (key) dependency

user1 → name1

user2 → name2

“Bad” (non-key) dependencies

Not really ok → But doesn’t violate any functional dependency!

It’s a **design anomaly** caused by the “improper” dependencies

Functional dependencies in a “bad” design

Follows

user1	name1	user2	name2	date
6	James	24	Amelia	‘2020-10-02’
6		12	David	‘2020-10-02’
81	Nicholas	12		‘2020-10-02’
12	Laura	73		‘2020-10-02’
12		73	Catherine	

$\{ \text{user1}, \text{user2} \} \rightarrow \text{date}$

$\text{user1} \rightarrow \text{name1}$

$\text{user2} \rightarrow \text{name2}$

Can you fill in the blanks?

Functional dependencies in a “bad” design

Follows

	user1	name1	user2	name2	date
Row $x \rightarrow$	6	James	24	Amelia	‘2020-10-02’
Row $y \rightarrow$	6	f(user1)	12	David	‘2020-10-02’
	81	Nicholas	12		‘2020-10-02’
	12	Laura	73		‘2020-10-02’
	12		73	Catherine	

{ user1, user2 } $\rightarrow$ date

user1 $\rightarrow$ name1

user2 $\rightarrow$ name2

Can you fill in the blanks?

$x.\text{user1} = y.\text{user1} \Rightarrow$ we should have $x.\text{name1} = y.\text{name1}$

From now on we are going to use new notation and terminology: the relational model

It expresses essentially “the same things” as table terminology but in a mathematical language

It emphasizes the formality behind DB concepts, and makes “shorter sentences”

Let’s open a brief parenthesis to introduce…
the relational model (see “4-relational-model.pdf”)

Functional dependency definition in table terminology

Given a table schema $T(a, b, \dots)$

We say that a column b functionally depends on another column a if:

$$t_1.a = t_2.a \Rightarrow t_1.b = t_2.b$$

for any two rows t_1, t_2 in any state of table T

We express this as $a \rightarrow b$, which we call a functional dependency

Functional dependencies can involve sets of columns: $\{a, b\} \rightarrow \{c, d\}$

Functional dependency definition in **relational model** terminology

Given a **relational schema** $T(a, b, \dots)$

We say that an **attribute** b functionally depends on another **attribute** a if:

$$t_1.a = t_2.a \Rightarrow t_1.b = t_2.b$$

for any two **tuples** $t_1, t_2 \in \boldsymbol{r(T)}$

We express this as $a \rightarrow b$, which we call a functional dependency

Functional dependencies can involve sets of **attributes**: $\{a, b\} \rightarrow \{c, d\}$

Based on the concept of **functional dependency**

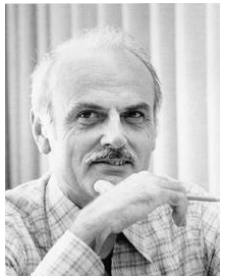
we can now define properties and levels

of desirable schema design: **normal forms**

Normal forms

- Satisfying them reduces update anomalies and improves the properties of the design (robustness, ease of evolution, etc.)
- They are incremental
  - If the nth form is satisfied, then the (n-1)-th is satisfied
- 1st, 2nd, 3rd and BCNF normal forms
  - 2nd, 3rd and BCNF are defined in terms of functional dependencies
  - They do not completely avoid all possibility of update anomalies, but they reduce them to very exceptional cases in practice
- 4th, 5th and 6th normal forms
  - They progressively avoid more and more update anomalies

Boyce-Codd normal form (BCNF)



Edgard F. Codd

Raymond F. Boyce

- Def: A schema R is BCNF if for all non-trivial dependency X $\rightarrow$ Y, we have that X is a superkey of R
- In other words, there cannot be any other dependency than the dependencies on superkeys

"*The key, the whole key, and nothing but the key –so help me Codd*"

Boyce-Codd normal form: example 1

Flight (<u>nflight</u>, origin, destination, time)

- Keys: <u>nflight</u>
- Dependencies: <u>nflight</u> $\rightarrow$ { origin, destination, time } } *Ok BCNF*

Passenger (<u>pid</u>, name)

- Keys: <u>pid</u>
- Dependencies: <u>pid</u> $\rightarrow$ name } *Ok BCNF*

Ticket (<u>pid</u>, <u>nflight</u>, <u>date</u>)

- Keys: { <u>pid</u>, <u>nflight</u>, <u>date</u> }
- Dependencies: $\varnothing$ } *Ok BCNF*

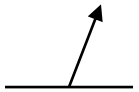
Normal form Boyce-Codd: example 2

Address (<u>street</u>, <u>number</u>, <u>floor</u>, municipality, province, <u>postalcode</u>)

- Examples
  - Address (‘Calle Cervantes’, 2, 2, ‘Castejón’, ‘Navarra’, 31590)
  - Address (‘Calle Cervantes’, 2, 1, ‘Castejón’, ‘Navarra’, 31590)
  - Address (‘Calle Cervantes’, 2, 1, ‘Castejón’, ‘Cuenca’, 16856)

3rd normal form (3NF)

$X \rightarrow A$ is trivial if $A \subset X$



- Def: A schema is 3NF if for all non-trivial dependency $X \rightarrow A$,
  either X is a superkey, or is a prime attribute

- Def: A schema attribute is **prime** iff it is part of some key of the schema

- In other words, an attribute should not depend on anything but a superkey,
  except perhaps the attributes that are part of some key

- The purpose is to admit "rare" exceptions to BCNF that are impossible
  to decompose without loss of dependencies

3rd normal form: example 1

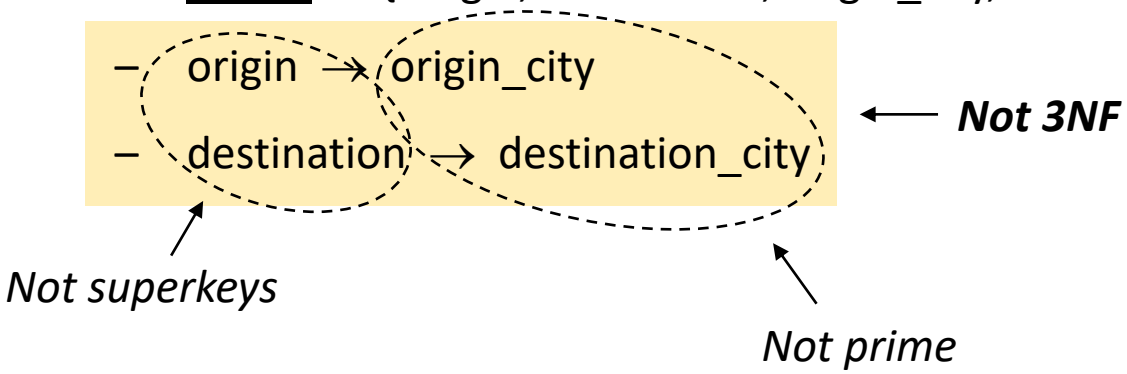
Address (<u>street</u>, <u>number</u>, <u>floor</u>, municipality, province, <u>postalcode</u>)

- Examples
  - Address (‘Calle Cervantes’, 2, 2, ‘Castejón’, ‘Navarra’, 31590)
  - Address (‘Calle Cervantes’, 2, 1, ‘Castejón’, ‘Navarra’, 31590)
  - Address (‘Calle Cervantes’, 2, 1, ‘Castejón’, ‘Cuenca’, 16856)

3rd normal form: example 2

Flight (<u>nflight</u>, origin, destination, origin_city, destination_city, time)

- Examples
  - Flight (123, 'CDG', 'LHR', 'Paris', 'London', '11:35:00')
  - Flight (456, 'ORY', 'LGW', 'Paris', 'London', '15:20:00')
- Keys
  - <u>nflight</u>
- Dependencies
  - <u>nflight</u> → { origin, destination, origin_city, destination_city, time }
  - origin → origin_city
  - destination → destination_city



2^{nd} normal form (2NF)

- Def: A schema R is 2NF if all non-prime attribute of R has a full functional dependency on the keys of R

- Def: A functional dependency $X \rightarrow Y$ is **full** if we cannot spare any of the attributes in X in the dependency, that is, $X - \{A\} \not\rightarrow Y, \forall A \in X$

- In other words, the attributes depend on the whole key; only the attributes of a key may depend on parts of a key

- The purpose is to admit denormalization for the sake of query efficiency, but avoiding to merge entities in a single table –avoid undermining the role of keys

2nd normal form: example 1

Flight (<u>nflight</u>, origin, destination, origin_city, destination_city, time)

- Examples
  - Flight (123, 'CDG', 'LHR', 'Paris', 'London', '11:35:00')
  - Flight (456, 'ORY', 'LGW', 'Paris', 'London', '15:20:00')
- Keys
  - <u>nflight</u>
- Dependencies
  - <u>nflight</u> → { origin, destination, origin_city, destination_city, time }
  - origin → origin_city
  - destination → destination_city

Ok 2NF

Not 3NF

Not superkeys

Not prime

2nd normal form: example 2

Ticket (name, pid, nflight, origin, destination, time, date, price)

- Examples
  - Ticket ('Susan', 165467, 123, 'MAD', 'LAX', '16:25:00', '2011-10-24', 620)
  - Ticket ('Peter', 165467, 123, 'MAD', 'LAX', '16:25:00', '2011-10-24', 620)
  - Ticket ('Peter', 165467, 123, 'MAD', 'LAX', '16:25:00', '2011-11-18', 620)
- Keys...
  - { pid, nflight, date } /* *Assuming a passenger only books tickets* */
  - { pid, date, time } /* *for her own flights* */
- Dependencies...
  - pid → name
  - nflight → { origin, destination }

  ← ***Not 2NF***
  - nflight → time /* *time is a prime attribute* */
  - { pid, nflight, date } → price
  - { pid, date, time } → { nflight, origin, destination, price }

  Ok

A last couple of examples…

- If all keys have a single attribute, ¿do we know something of the normal form of the schema already?
- ¿What is the minimum normal form of a schema where the combination of all the attributes is a key?

1st normal form (1NF)

- Def: in a 1NF schema…
  - Attributes are atomic and univalued
  - Attribute names are unique
  - There are no duplicate tuples (consequence: every schema has a key)
  - The order of tuples and attributes is arbitrary
- It is considered an inherent part of the relational model
- Although…
  - SQL only strictly satisfies the first of the above conditions
  - Alternatives to the relational model are studied, such as the nested relational model, that admit relations as attribute values

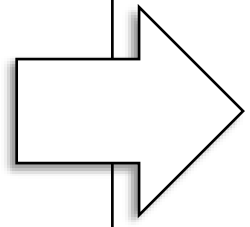
Boyce-Codd normal form (BCNF)

BCNF does not fully remove all update anomalies → 4th, 5th, 6th NF

brand	medicine	uses
Gelocatil	Paracetamol	Fever
Gelocatil	Paracetamol	Pain
Gelocatil	Paracetamol	Cefalea
Tylenol	Paracetamol	Fever
Tylenol	Paracetamol	Pain
Tylenol	Paracetamol	Cefalea
Tylenol	Codeine	Cough
Tylenol	Codeine	Pain

Redundancies…

- The schema is BCNF
- But it contains redundancies
- Update anomalies: modify / add / remove uses of a medicine, etc.



Better design…

brand	medicine
Gelocatil	Paracetamol
Tylenol	Paracetamol
Tylenol	Codeine

medicine	uses
Paracetamol	Fever
Paracetamol	Pain
Paracetamol	Cefalea
Codeine	Cough
Codeine	Pain

Summary

<table>
<tr><td colspan="2" rowspan="2">Given X → Y…</td><td colspan="3">X superkey</td></tr>
<tr><td rowspan="2">Yes</td><td colspan="2">No</td></tr>
<tr><td colspan="2">X prime</td></tr>
<tr><td rowspan="2">Y prime</td><td>Yes</td><td rowspan="2">BCNF</td><td colspan="2">3NF</td></tr>
<tr><td>No</td><td>1NF</td><td>2NF</td></tr>
</table>

Relational schema normalization

There are two ways to obtain schemas in a certain normal form

1. We design normalized schemas from the beginning
   - This results in general naturally, for instance, when we convert an ER model into relational schemas by systematic steps
2. We start from a low normal form design
   - For instance, a legacy design, or we did not know any better initially
   - We normalize the schema $\rightarrow$ this consists in decomposing the initial schema into several schemas in the desired normal form
   - But not in no matter what way…

Relational schema decomposition

A decomposition of a schema R is a mapping $\mathrm{R} \rightarrow \{\mathrm{R}_1, \ldots, \mathrm{R}_n\}$

It is the basis of normalization

It is desirable that the decomposition satisfies the following properties:

- Attribute preservation
  - That is $\bigcup_{i=1}^{n} \mathrm{R}_i = \mathrm{R}$
- Dependency preservation
  - Let F be the set of functional dependencies of R
  - Let $\pi_{\mathrm{R}_i}(\mathrm{F})$ be the subset of dependencies of F that involve attributes of R_i
  - We should have that F is equivalent to $\bigcup_{i=1}^{n} \pi_{\mathrm{R}_i}(\mathrm{F})$ } *To be seen soon…*

  In other words, the union of the dependencies of tables R_i should be equivalent to the dependencies of the original table
- Lossless join (or non-additive join –loss of information, not of tuples)
  - For all state r of R we must have $\pi_{\mathrm{R}_1}(r) \bowtie \pi_{\mathrm{R}_2}(r) \bowtie \cdots \bowtie \pi_{\mathrm{R}_n}(r) = r$, where:

    $\pi_{\mathrm{R}_i}(r)$ is the projection of the tuples of r on the attributes of R_i

    $\bowtie$ is natural join

    In other words: the join of tables R_i is the original table R

    } *Operations of relational alge-bra (soon later…)*
  - Avoids spurious tuples when applying natural join

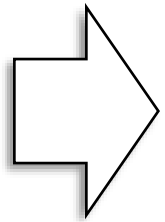
Schema decomposition

When decomposing tables…

- Don’t lose attributes
- Join of tables $\rightarrow$ original table
- Don’t loose dependencies (if possible)

Enrollment

sid	name	telephone	course
12345	Jane	123456789	17824
12345	Jane	123456789	17825
67890	David	321654987	17826
67890	David	321654987	17827
89456	Jane	755326284	17835
89456	Jane	755326284	17838
⋮	⋮	⋮	⋮



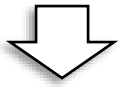
Student

sid	name	telephone
12345	Jane	123456789
67890	David	321654987
89456	Jane	755326284
⋮	⋮	⋮

Enrollment

name	course
Jane	17824
Jane	17825
David	17826
David	17827
Jane	17835
Jane	17838
⋮	⋮

SELECT * FROM Student
NATURAL JOIN Matricula



Spurious tuples: example

sid	name	Telephone	course
12345	Jane	123456789	17824
12345	Jane	123456789	17825
12345	Jane	123456789	17835
12345	Jane	123456789	17838
67890	David	321654987	17826
67890	David	321654987	17827
89456	Jane	755326284	17835
89456	Jane	755326284	17838
89456	Jane	755326284	17824
89456	Jane	755326284	17825
⋮	⋮	⋮	⋮

Spurious tuples

Normalization algorithms

- Check dependency preservation in schema decomposition
- Check lossless join in schema decomposition
- Check that a set of attributes is a superkey
- Satisfaction of 3NF and BCNF by schemas
- Decomposition of relations into 3NF, BCNF
  - It is always possible to decompose into 2NF and 3NF without dependency loss
  - BCNF may not always be possible without dependency loss

3NF normalization

3NF (R, F)

What's this?

D := ∅

G := Minimal cover of F

for $X \rightarrow Y \in G$

Add to D the schema $X \cup \{ A_1, A_2, \ldots, A_n \}$

where $X \rightarrow A_i$ are all the dependencies on X in G

If no schema in D contains a key of R, add to D a schema with the key of R

Remove all redundant schemas in D (schemas contained in another schema)

The resulting decomposition has lossless join, and preserves dependencies

3NF normalization (cont)

In other words…

1. Starting from a minimal cover
2. Create a schema with the attributes of all dependencies with the same "left hand side" (which will be the key of the table)
3. If no schema has resulted that contains the full original key, create that table
4. Remove redundant schemas

Minimal cover

- Def: A minimal cover of a set of dependencies F is a minimal set equivalent to F
- Def: A set of dependencies F is minimal if:
  - The right hand side of all dependencies is a single attribute
  - If we remove a dependency, we obtain a set that is not equivalent to F
  - If we remove an attribute from the right hand side of a dependency, we obtain a set that is not equivalent to F

  In other words: dependencies in canonical form and without redundancies
- Def: Two sets of dependencies are equivalent if all the dependencies of one can be inferred from dependencies of the other and vice versa

Inference between dependencies

Reflexivity:	$Y \subset X$	$\Rightarrow$	$X \rightarrow Y$ (trivial dependency)
Augmentation:	$X \rightarrow Y$	$\Rightarrow$	$XZ \rightarrow YZ$
Transitivity:	$\{ X \rightarrow Y, Y \rightarrow Z \}$	$\Rightarrow$	$X \rightarrow Z$
Projection:	$X \rightarrow YZ$	$\Rightarrow$	$X \rightarrow Y$ ([
Additivity:	$\{ X \rightarrow Y, X \rightarrow Z \}$	$\Rightarrow$	$X \rightarrow YZ$ (union)
Pseudo-transitivity:	$\{ X \rightarrow Y, WY \rightarrow Z \}$	$\Rightarrow$	$WX \rightarrow Z$
Note that que:	$\{ X \rightarrow Y, Z \rightarrow W \}$	$\Rightarrow$	$XZ \rightarrow YW$
	$X \rightarrow Z$	$\Rightarrow$	$XY \rightarrow Z$

Rules ("axioms") of Armstrong, complete

Minimal cover (cont)

Minimal cover (E)

F := E

Replace al dependencies $X \rightarrow \{ Y_1, \ldots, Y_n \}$ in F

by $X \rightarrow Y_1, \ldots, X \rightarrow Y_n$

for $X \rightarrow Y \in F$ do

for $A \in X$ do

if $F - \{ X \rightarrow Y \} \cup \{ (X - \{A\}) \rightarrow Y \}$ is equivalent to F

then $F := F - \{ X \rightarrow Y \} \cup \{ (X - \{A\}) \rightarrow Y \}$

for $X \rightarrow Y \in F$ do

if $F - \{ X \rightarrow Y \}$ is equivalent to F

then $F := F - \{ X \rightarrow Y \}$

return F

Minimal cover (cont)

In other words…

1. Decompose into dependencies with individual attributes on the right hand side
2. Remove unnecessary attributes on left hand side
3. Remove dependencies that can be inferred from others

BCNF normalization

BCNF (R, F)

D := {R}

while D contains a non BCNF schema

Q := pick a non BCNF schema in D

$X \rightarrow Y$:= chose a non BCNF dependency of F in Q

Replace Q in D by $(Q - Y)$, $(X \cup Y)$

The algorithm generates a lossless decomposition,
but does not guarantee preservation of all dependencies

BCNF normalization (cont)

In other words…

Dependencies can be lost here

1. Take to table apart all non BCNF dependencies of the original schema, but removing from the latter all the “right hand side” of dependencies
2. Repeat the process on all the schemas that get produced

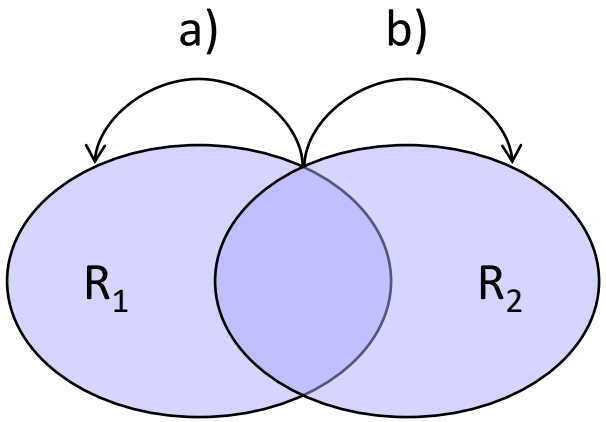
Lossless join

Simple test:

A binary decomposition $\{R_1, R_2\}$ of a relation R has lossless join with respect to a set of dependencies F if as a consequence of F:

Either a) $R_1 \cap R_2$ is a superkey in R_1 (and acts as foreign key in R_2)

Or b) $R_1 \cap R_2$ is a superkey in R_2 (and acts as foreign key in R_1)



Other design criteria

Besides normalization… (motivated by update anomalies)

- Avoid NULL values
  - They create problems in operations involving comparisons, counts or sums
  - The ratio of NULLs in an attribute can be a criterion for taking the attribute to a separate relation
- Schema semantics
  - The ease of explanation is an informal measure of design quality
  - E.g. a schema that aggregates several real world entities can be more confusing
- Efficiency and denormalization
  - Denormalized tables can be more efficient for some queries
  - For efficiency it sometimes pays off to merge or not decompose certain tables: give up on space and change robustness, in exchange for faster queries (sparing joins)
  - It depends on the query frequency, table size and value frequency