Design Theory

1. Keys & FDs
2. The Normal Forms
3. Reasoning with FDs
4. Normalization

Parke Godfrey

EECS-3421A | Fall 2020 | class site

1 . 1

Versions

initial:	*2020 January 28*
last modified:	*2020 September 30*

1.2

Acknowledgments

Thanks to

- *Jeffrey D. Ullman*
  for initial slidedeck
- *Jarek Szlichta*
  for the slidedeck with significant refinements on which this is derived
- *Wenxiao Fu*
  for refinements

1 . 3

Printable version of talk

- Follow this link:
  Design Theory [to **pdf**].
- Then *print* to get a **PDF**.

(*This only works correctly in **Chrome** or **Chromium**.*)

1 . 4

1. Keys & FDs

FDs: *functional dependencies*

functional dependency

- relationship between 2 attributes.
- typically between primary key and a non-key attribute within a table.

2.1

Notation

Let us first establish the *terminology* we shall be using.

> We shall now be quite formal. But this formality will pay off.

2.2

functionally determines

$\mathcal{R}$ → set of attribute

$\mathbf{R}$ → table

Let $\mathcal{R}$ be the *set* of attr's of table $\mathbf{R}$.

If subset of attr's $\mathcal{K} \subseteq \mathcal{R}$ is "the" *key* of $\mathbf{R}$, then there is *at most* one tuple with given values for the attr's in $\mathcal{K}$ in (any instance of) table $\mathbf{R}$.

Another way to think of this is that, given values for $\mathcal{K}$, there is a distinct value for each attr. in $\mathcal{R} - \mathcal{K}$.

In this case, we say that $\mathcal{K}$ *functionally determines* $\mathcal{R}$. We denote this by

$$\mathcal{K} \mapsto \mathcal{R}$$

Key

set of attributes

2.3

How is this like a *function*?

Why do we call this *functional*? → it acts like a function

It *is* like a function!

- One puts a value *into* the function, which maps to *a* value out.
- It is a *partial* function, not *total*; not *every* value as input maps to an output value.
- Where is this function encoded?
  - Not by a mathematical formula, as, perhaps, we are used to.
  - It is encoded in the *data* in the database.

2.4

keys & superkeys

superkey

superset of key attributes that determine other attributes

Given $\mathcal{R}$ — the set of all attr's of table $\mathbf{R}$ — we call *any* subset $\mathcal{S} \subseteq \mathcal{R}$ such that $\mathcal{S} \mapsto \mathcal{R}$ a *superkey* of table $\mathbf{R}$.

---

key

If no *proper* subset of a superkey $\mathcal{K}$ for $\mathbf{R}$ is *also* a superkey for $\mathbf{R}$ — that is, $\neg\exists \mathcal{J} \subset \mathcal{K}.\ \mathcal{J} \mapsto \mathcal{R}$ — then we call $\mathcal{K}$ a *key* of table $\mathbf{R}$.

1

2.5

functional dependencies

We might happen to know that, in some domain, $\mathcal{X} \mapsto \mathcal{Y}$ must be *true* (i.e., "holds"), where $\mathcal{X} \cup \mathcal{Y} \subset \mathcal{R}$, but $\mathcal{X} \not\mapsto \mathcal{R}$; that is, $\exists \mathrm{A} \in \mathcal{R} - (\mathcal{X} \cup \mathcal{Y}).\ \mathcal{X} \not\mapsto \{\mathrm{A}\}$.

Thus, $\mathcal{X}$ is *not* a superkey of $\mathbf{R}$! But such things as $\mathcal{X} \mapsto \mathcal{Y}$ will be important.

We call $\mathcal{X} \mapsto \mathcal{Y}$ a *functional dependency* ("FD").

If $\mathcal{X} \cup \mathcal{Y} = \mathcal{R}$, call an FD a *superkey FD if* its left-hand side (LHS) is a *superkey*; call it a *key FD if* its LHS is a *key*.

Not all FDs are superkey FDs, of course!

2.6

Why do *we* need FDs?

When we have keys already!

Because some FDs that hold in our domain may not be superkey FDs for our schema.

When this is the case, the schema can violate our principle of *single source of truth*. (*Deletion* and *insert* / *update* anamolies would be possible.)

By *reasoning* with FDs, we can vet whether our schema had good logical properties.

And when our schema does not, this will provide us tools for refactoring the schema to achieve the good properties.

2.7

Canonical Form

splitting right-hand sides

Consider $\mathcal{X} \mapsto \mathcal{Y}$ where $\mathcal{Y} = \{\mathrm{Y}_0, \ldots, \mathrm{Y}_{\mathrm{k}-1}\}$. Then that FD is equivalent to the set of FDs

$$\forall i \in \{0, \ldots, k-1\}.\ \mathcal{X} \mapsto \{\mathrm{Y}_\mathrm{i}\}$$

We generally express FD's with *singleton* right-hand sides.

There is no *splitting rule* for left-hand sides! That would be incorrect.

2.8

Shorthand for FDs

As shorthand, instead of using “{”'s and “}”'s everywhere, when we are using single letters for attr's, we just munge them together.

E.g., we write $\{A, B, C\} \mapsto \{D, E\}$ as $ABC \mapsto DE$.

2.9

Example: Drinker & FDs

Drinker(name, address, beer, manf, favBeer)

Here, we mean the drinker (name) likes that beer, and it is manufactured by manf.

Reasonable FDs to assert:

- name ↦ addr, favBeer
  equivalent to
  - name ↦ addr
  - name ↦ favBeer

  } Splitting name ↦ addr, favBeer
- beer ↦ manf

Primary Key
name, beer

3.1

Example: key of Drinker

$\{\text{name}, \text{beer}\}$ is a *key* of **Drinker** because neither $\{\text{name}\}$ nor $\{\text{beer}\}$ is a superkey.

- $\text{name} \not\to \text{manf}$
- $\text{beer} \not\to \text{addr}$

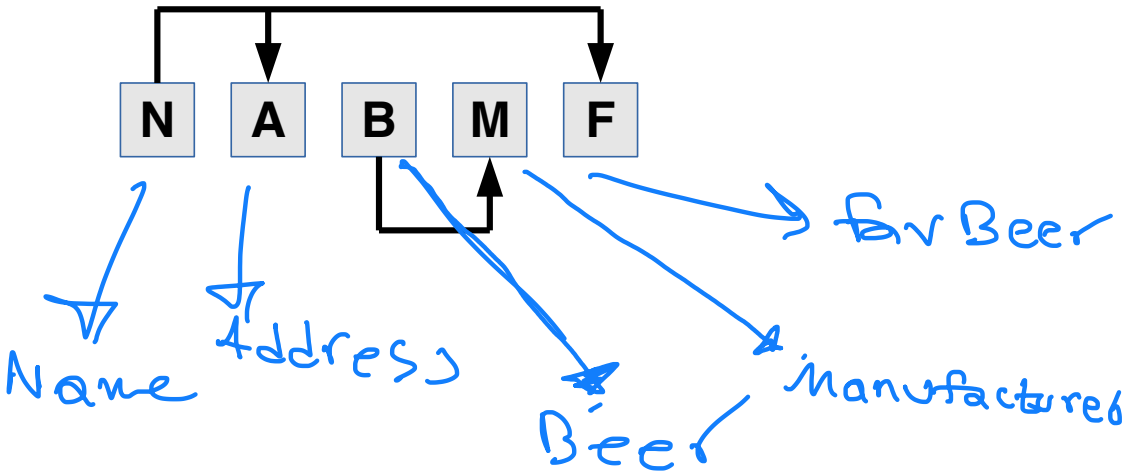
In this case, there are no other keys.

But there are lots of superkeys! Namely, any superset of $\{\text{name}, \text{beer}\}$.

3.2

Visualizing

We sometimes draw out FDs to see what is going on.



3.3

Where do FDs come from?

- The designer *presribes* them by naming keys.
- From real-world "constraints" that the designer knows and *prescribes*.

  E.g., "no two classes can meet in the same room at the same time"

$$\text{hour}, \text{room} \mapsto \text{class}$$

Because we may have FDs *in addition to* the prescribed key FDs, additional key FDs might exist.
We may have to *assert* our FDs, then *deduce* the keys by systematic exploration!

4

Problem: non-key FDs

Non-key FDs are problematic, because they allow for the potential of *anomalies*.

---

our game: Design to have only superkey FDs.

5.1

Anomalies / redundancies

- **update anomaly**: one occurrence of a fact is changed, but not all occurrences are.
- **deletion anomaly**: a valid fact is lost when a tuple is deleted.

The *goal* of relational schema design is to avoid such anomalies and redundancy.

Non-key FDs cause these problems because these violate our “single source of truth” mandate.

5.2

Upto here on Oct-1

Example: anomalies

- **deletion anomaly**.
  - No drinker likes the beer *Bud*.
  - Thus, no tuple appears in **Drinker** with $\text{beer} = \text{'Bud'}$.
  - Then we do not have the information that $\text{beer} = \text{'Bud'}$ and $\text{manf} = \text{'Anheuser Busch'}$.
- **insertion / update anomaly**.
  - Say there is a tuple in **Drinker** with $\text{beer} = \text{'Bud'}$ and $\text{manf} = \text{'Anheuser Busch'}$ (with, say, $\text{name} = \text{'Jeff'}$). *correct*
  - Someone accidentally adds another tuple later with $\text{beer} = \text{'Bud'}$ and $\text{manf} = \text{'Labatt'}$ (with, say, $\text{name} = \text{'Franck'}$). *incorrect*
  - **Note**. Our key of $\{\text{name}, \text{beer}\}$ has *not* been violated.
  - Who manufactures *Bud*?

5 . 3

2. The Normal Forms

Process of minimizing redundancy from a relation or set of relation

6.1

First pass

What the normal forms do

1. **1NF**: Every table (relation) has a key *prescribed*. (*Trivial* to achieve. Why?) atomic and every table key
2. **2NF**: Every table is in 1NF *and* no table has a *partial-key dependency*.
3. **3NF**: Every table is in 2NF *and* no table has a *transitive dependency*.
4. **BCNF**: Every table is in 3NF *and* no table has a *back dependency*.

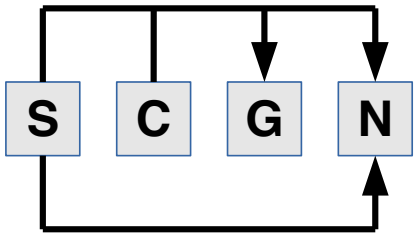
If we can achieve BCNF, we should be good to go!

Boyce-Codd normal form

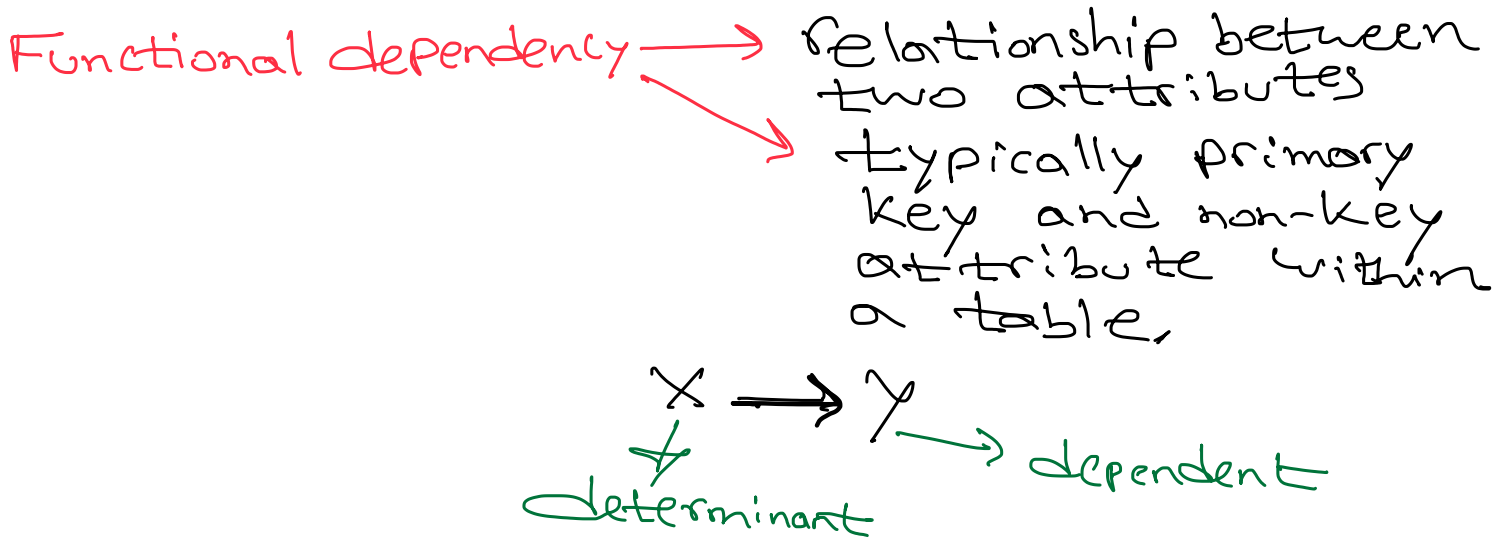
6.2

partial-key dependencies (¬ 2NF)

Consider **Enrol**(s#, c#, grade, student_name) (SCGN).

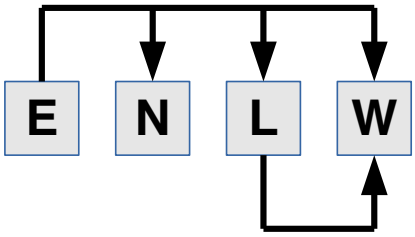


- "The" key FD is clearly $SC \mapsto GN$.
- We also have the non-key FD $S \mapsto N$.
  - We call this a *partial-key* dependency because its LHS is a proper subset of *a* key.
    E.g., $\{S\} \subset \{S, C\}$.



transitive dependencies (¬ 3NF)

Consider $\mathbf{Emp}(\text{emp\#}, \text{name}, \text{level}, \text{wage})$ (ENLW).



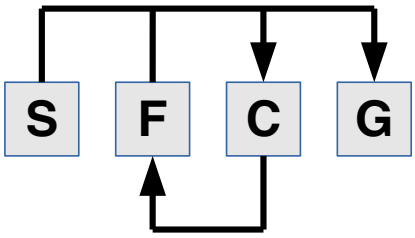
- "The" key FD is $\mathrm{E} \mapsto \mathrm{NLW}$.
- We also have the non-key FD $\mathrm{L} \mapsto \mathrm{W}$.
  - We call this a *transitive* dependency because its LHS and RHS are not subsets of *any* keys.
    E.g., $\{\mathrm{L}\} \not\subseteq \{\mathrm{E}\}$ and $\{\mathrm{W}\} \not\subseteq \{\mathrm{E}\}$.

LHS RHS

6.4

back dependencies (¬ BCNF)

Consider $\mathbf{Enrol_2}$(st#, fac#, class#, grade) (SFCG).



- "The" key FD is $\mathrm{SF} \mapsto \mathrm{CG}$.
- We also have the non-key FD $\mathrm{C} \mapsto \mathrm{F}$.
  - We call this a *back* dependency because its LHS *seems* to not be a subset of *a* key but its RHS is part of a key.
    E.g., $\{\mathrm{C}\} \subseteq \{\mathrm{SF}\}$.

6.5

SC ⟼ FG key!!

Second pass

Formally defining things

But your “definitions” above seem *fuzzy*!
“*Seems* not to be a subset of *a* key?!”

The examples above give you a *feel* for what is going on.

Why are they not *formal* definitions?!

We did not say what the situation is when

- the LHS or RHS of a non-key FD overlaps with a key, *or*
- there is *more* than one key.

7.1

What were the keys in $\mathbf{Enrol_2}$?

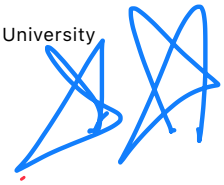
Well, $\{\mathrm{S}, \mathrm{F}\}$. That was stated!

But also $\{\mathrm{S}, \mathrm{C}\}$!

But then… wouldn't $\mathrm{C} \mapsto \mathrm{F}$ be a partial-key dependency?!

7.2

Formal definitions



- **1NF**: Each attr. is an elementary type. A key is defined for each relation.
- **2NF**: Whenever $\mathcal{X} \mapsto \mathrm{A}$ holds in $\mathbf{R}$ and $\mathrm{A} \notin \mathcal{X}$,
  - A is *prime*, or
  - $\mathcal{X}$ is not a proper subset of *any* key of $\mathbf{R}$.
- **3NF**: Whenever $\mathcal{X} \mapsto \mathrm{A}$ holds in $\mathbf{R}$ and $\mathrm{A} \notin \mathcal{X}$,
  - A is *prime*, or
  - $\mathcal{X}$ is a superkey of $\mathbf{R}$.
- **BCNF**: Whenever $\mathcal{X} \mapsto \mathrm{A}$ holds in $\mathbf{R}$ and $\mathrm{A} \notin \mathcal{X}$,
  - $\mathcal{X}$ is a superkey of $\mathbf{R}$.

An attr. is *prime* if it is part of *any* key of $\mathbf{R}$. E.g., $\mathrm{name} \in \{\mathrm{name}, \mathrm{beer}\}$ in **Drinker**.

7.3

3. Reasoning with FDs

Some FDs that hold for a relation may be *implicit*.

That is, an implicit FD may logically follow from the set of known FDs.

We will need to *infer* the implicit ones. For example, we need to know a relation's keys to test whether it is in BCNF.

Is this *hard*? Well … *yes* and *no*.

8.1

Why is it *hard*? (I)

How many different keys are possible for a rel'n that has *one* key?

Exponential!

Consider a rel'n with n attr's. Then, there are 2^n possible different keys.

How many possible keys of length k?

$$\binom{n}{k}$$

8.2

Why is it *hard*? (II)

How *many* keys might a rel'n have (*simultaneously*)?

There can be lots of FDs! And lots of *key* FDs, too.

Consider I know that $A_1 \mapsto A_2$, $A_2 \mapsto A_1$, …, $Z_1 \mapsto Z_2$, $Z_2 \mapsto Z_1$.

There are 2^{26} keys!

A rel'n can be an *exponential* number of *key* FDs.

8.3

What is easy?

From a set of attr's $\mathcal{X}$, I can find the *closure*, $\mathcal{X}^+$, of that set of attr's easily.

(In fact, there is a *linear* runtime algorithm known for this!)

O(n)

8.4

FD axiomatization

A *sound* and *complete* set of axioms for FDs is as follows.

1. *Reflexivity*. $\mathcal{X} \mapsto \mathcal{Y}$ if $\mathcal{Y} \subseteq \mathcal{X}$.
   - These are called *trivial* FDs.
2. *Augmentation*. If $\mathcal{X} \mapsto \mathcal{Y}$ then $\mathcal{X} \cup \mathcal{Z} \mapsto \mathcal{Y} \cup \mathcal{Z}$ for any $\mathcal{Z}$.
3. *Transitivity*. If $\mathcal{X} \mapsto \mathcal{Y}$ and $\mathcal{Y} \mapsto \mathcal{Z}$ then $\mathcal{X} \mapsto \mathcal{Z}$.

9.1

Computing *closure* $^+$

Computing the *closure* of a set of attr's $\mathcal{X}$ is just the *fixpoint* with respect to *transitivity* then.

Let $\mathcal{X}_0 = \mathcal{X}$.

Define $\mathcal{X}_{i+1} = \mathcal{X}_i \cup \bigcup_{\boldsymbol{\mathcal{F}}_i} \mathcal{Z}$.

where $\boldsymbol{\mathcal{F}}_i = \{\mathcal{Y} \mapsto \mathcal{Z} \mid \mathcal{Y} \subseteq \mathcal{X}_i \wedge \mathcal{Z} \nsubseteq \mathcal{X}_i\}$.

Then $\mathcal{X}^+ = \mathcal{X}_i$ for which $\boldsymbol{\mathcal{F}}_i = \{\}$.
Or, equivalently, $\mathcal{X}^+ = \mathcal{X}_i$ for which $\mathcal{X}_i = \mathcal{X}_{i+1}$.

9.2

Ever need to find *all* keys of a rel'n?

Unfortunately, we might have to.

For instance, to check a *schema* for meeting normal form.

See **Exercise #9** from *Exercises for Study* (and *with answers*).

4. Normalization

refactoring a schema

Goal: A "correct" schema in BCNF.
Or in 3NF, if that is not possible.

So…how to achieve BCNF (or 3NF)?

3rd Normal form

Schema refactoring → fixing a schema

10 . 1

Two approaches (approach for 3NF)

decomposition: a *top-down* approach

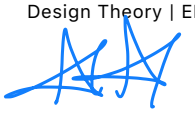
- Start with our schema and *refine* it.

synthesis: a *bottom-up* approach

- Start with the prescribed FDs and *create* a schema from them.

decomposition → top-down
Synthesis → bottom-up

10.2



Decomposition

Method Sketch

1. Find a *problematic* FD; i.e., one that breaks xNF.
2. Make the problematic FD “go away”. How?
   - Let the FD be (a) key of its own rel'n.
   - That is, split the non-xNF rel'n into *two* rel'ns in a *lossless* way.
3. Repeat until no more problematic FDs!

See Exercise #16 from *Exercises for Study* (and *with answers*).

11 . 1

Two goals (of 3NF)

1. ***lossless* decomposition**
   - We must ensure that the *final* schema can "reproduce" our *original* schema. That nothing is *lost*. (final schema is logically equivalent to original schema)
2. **dependency preservation**
   - *All* of our FDs are ensured by the *final* schema (by rel'n's keys).

(all functional dependencies are protected)

11 . 2

Lossless join decomposition

Say we have a rel'n schema $\mathbf{R}$ of ABCDE and an FD $\mathrm{B} \mapsto \mathrm{E}$ that violates BCNF for $\mathbf{R}$.

We can break $\mathbf{R}$ into *two* rel'ns:

1. BE
   - This is just the “FD” itself!
   - Its key can be B, the LHS of the FD.
2. ABCD
   - And the rest of what is left over from $\mathbf{R}$
   - …repeating the LHS of the FD!
     This will serve as a foreign key from 2) to 1).
   - The key can be whatever is key for ABCD.

Lossless join decomposition (general)

Given $\mathbf{R}$ and an FD $\mathcal{X} \mapsto \mathcal{Y}$ violating $\mathbf{R}$ — assume that $\mathcal{X} \mapsto \mathcal{Y}$ is non-trivial (that is, $\mathcal{X} \cap \mathcal{Y} = \{\}$) — replace $\mathbf{R}$ by two rel'ns:

1. $\mathcal{X} \cup \mathcal{Y}$
2. $\mathcal{R} - \mathcal{Y}$

What goes wrong if not *lossless*?

We cannot reproduce the “database” with respect to the original *schema* from our *decomposed* schema.

That is, if we “joined” our two decomposed rel'ns, we might not recover the original rel'n.

→ Origial database can't be reproduced

→ original relations can't be recovered.

11 . 5

Example of a bad decomposition

Consider ABC with no FDS and I break it into AB and BC. Let our table be

A	B	C
1	2	3
4	2	5

Example of a bad decomposition (2)

This decomposes into

A	B
1	2
4	2

Example of a bad decomposition (3)

But "joining" these back together ($AB \bowtie BC$) does *not* give us the same thing that was in $\mathbf{R}$!

A	B	C
1	2	3
1	2	5
4	2	3
4	2	5

The extra resulting tuples here from the "lossy" join are called *spurious*.

Problem:
→ we are gaining unnecessary information

But what about *dependency preservation*?

good news

- Lossless join decomposition steps can always get us eventually to BCNF! (first goal is preserved).

bad news

- the *resulting* schema may not be dependency preserving. (second goal isn't preserved)

See Exercise #17 from *Exercises for Study* (and *with answers*).

12 . 1

Decomposition

Revised Method

1. Find a *problematic* FD for a rel'n, decompose the rel'n losslessly with respect to the FD.
2. Repeat until no more problematic FDs!
3. Add back any FDs that are not covered in the *resulting* schema as rel'ns.

12.2

Can we always have BCNF?

The method above often works, but does not always.

What can go wrong? Some of the non-covered FDs that we add back in as rel'ns may not be in BCNF!

- Well, we could decompose an added, non-BCNF rel'n…
- But then the corresponding FD is not covered *again*!
- *Stuck*.

The result *does* arrive always to a 3NF, dependency preserving schema!

12.3

Is there just *one* lossless decomposition?

Of course not! The decomposition depends on the *order* of the decomposition steps we apply.

There may be an *exponential* number of lossless-join decompositions.

One of them might be BCNF and dependency preserving (after we add back in the non-covered)!

- But finding this one might be *hard*.
- And it might not even exist!

12 . 4

Synthesis Method

Find a *minimal basis* of the set of declared (explicit) FDs.

That's it!

13 . 1

Is polynomial!

It is *polynomial* to find *a* minimal basis!

The resulting schema *is* in 3NF and it *is* dependency preserving.

Note. There is no notion of “lossless” here; there is no “original” schema to compare against.

Finding *all* minimal bases, though, is *exponential*.

- One of them might be in BCNF too!
- But this is *NP-complete* to find.

13 . 2

Minimal basis

1. Throw away any FD that can be *derived* from the others (the remaining ones)

   Repeat until no such FD remains.

2. Throw away any attr. on the LHS of an FD *if* the resulting FD plus the others can *derive* the original FD.

   Repeat until no such FD remains.

See Exercise #18 from *Exercises for Study* (and *with answers*).

13 . 3