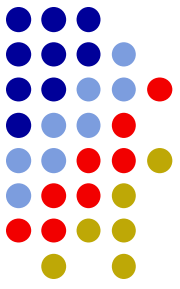


Material that was covered in depth

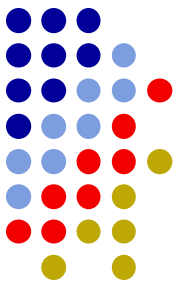
This will be on exercise 6 and test 2



- the discussion about keys
- the informal introduction to FDs
- “troublesome” FDs (the ones that are not implied by the keys)
- the FDs that are implied by the key(s) for a table
- the way you can determine the key for a table based on a set of FDs of the right form
- the decomposition algorithm (where the attributes in the troublesome FDs with the same LHS are “lifted” into a table of their own) were covered in depth.

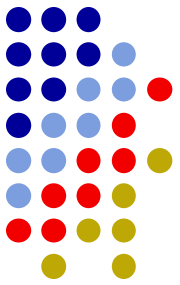
The exercises in those sections of the lecture were done in class.

Material that was covered in moderate depth



- The (historic) definition of 2NF, 3NF, and BCNF
- You don't need to memorize the names of these normal forms.
- For exercise 6/text 2, you do need to be able to recognize when a table is not in BCNF and be able to decompose it using the “lifting” decomposition algorithm. You should be able to do that for all three of the patterns shown – the pattern that violated 2NF, the pattern that violates 3NF and the pattern that violated BCNF (but not 3NF). Note: a given table may have multiple FDs that have several of these patterns.

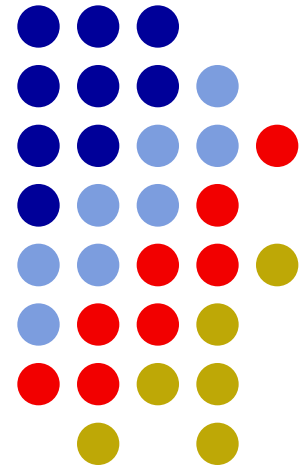
The material that was touched on

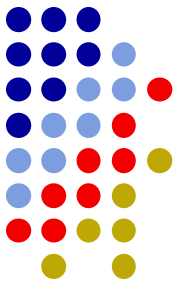


- The formal definition of FD (as a function)
- The formal definition of a key (where the attributes in a key must functionally determine all attributes in the table)
- The correctness of the “lifting” decomposition algorithm (i.e., the attributes in common must be a key for at least one of the two resulting tables)
- The description of how an FD can be lost
- An overview of how you can check to see if an FD is lost
- The set of inference rules that allow you to determine all the FDs implied by a given set of FDs
- The counterexample of how you can’t have BCNF and dependency preservation

Overview of Normalization based on FDs

Lois Delcambre
CS386/586
Winter 2013



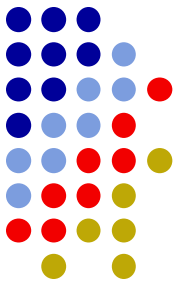


Definition of a Key for a Relation

A key is a **minimal** set of attributes in a relation whose values are guaranteed to uniquely identify rows in the relation.

- Two distinct rows have distinct key values
- (minimal) No subset of the fields that comprise a key is a key

If a key is not minimal, then there are extra attributes that you don't really need. Such a key is called a superkey.



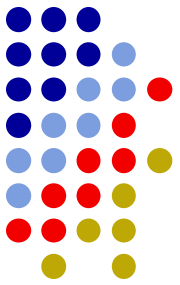
Quick Exercise

- Write down a table – with a number of attributes.
- Choose the key(s) for the table.
- Choose at least two superkeys for the table; use sample data to convince yourself that a superkey is a key (it uniquely identifies the rows) and that the extra attributes don't need to be in the superkey.

Do this exercise for a table where the key consists of one attribute.

Do this exercise for a table where the key has more than one attribute.

Notice ... only one value for each non-key attribute (for each key value)

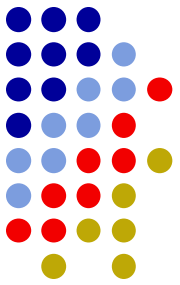


Employee	SSN	Name	Salary	Job-code
	111111111	John Smith	40,000	15
	123456789	Mary Smith	50,000	22
	123456789	Marie Jones	50,000	24

NOT allowed because SSN is key! →

Only one name (and one salary and one Job-code) for each row.

For one particular SSN value, **123-45-6789**, there is only **ONE name** because there is only one tuple and we assume that attributes values are atomic.



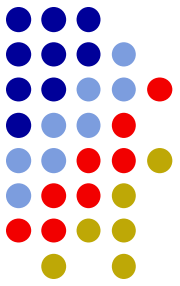
Notice what happens w/o a key

Student table with departments

<u>id</u>	name	dept-code	dept-name
101	Joe	CS	Computer Science
102	Wei	G	Geography
103	Mary	CS	Psychology

Nothing prevents CS from being listed with “Computer Science” as well as with “Psychology”; id as a key only prevents duplicate ids which means, for example, that 101 has only one name.

If you want each code to have just one value, use a table with code as key.



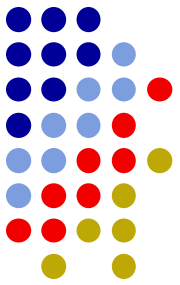
For each subject code, there is only one subject. (These codes are excerpted from the PSU Schedule of Classes.)

For example, the code “G” is used for Geology (and not for Geography for example).

You can **always** use a table, with the code and description as attributes, with the code as a key, to enforce this situation.

Subject-Lookup

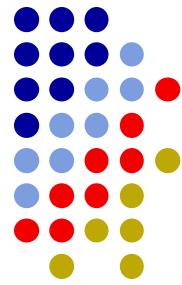
<u>Code</u>	Subject
CS	Computer Science
G	Geology
GEOG	Geography
HST	History
MTH	Mathematics
STAT	Statistics



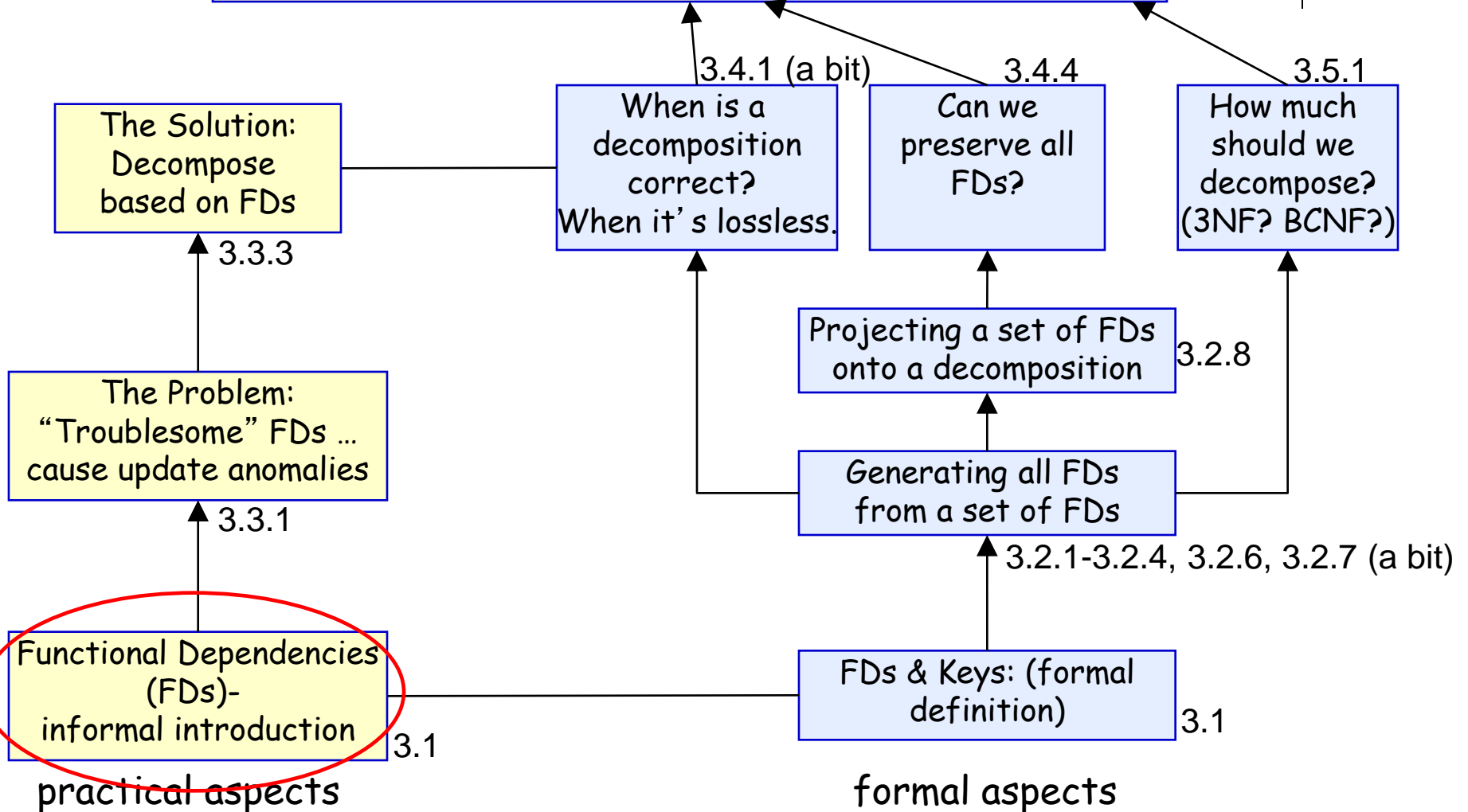
Quick Exercise

- Define a table with a key where there is information in the table (like the code and the subject) in the same table.
- Provide sample data that demonstrates that we can uphold the key but still have the problem shown (e.g., different department subjects for the same code).
- Show how the problem can be fixed if you introduce a new table for the codes and names.
- Such a new table is often called a lookup table; a foreign key (from the original table) enforces that the attribute has a valid value.

Normalization Strand Map

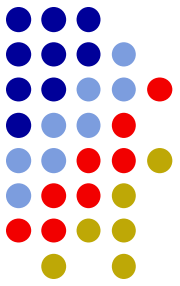


Goals for Normalization:
(1) Lossless (Correct) Required, (2) All tables in BCNF, and
(3) All FDs Preserved. Sometimes you must choose (2) vs. (3).



Functional Dependencies

Intuitive description

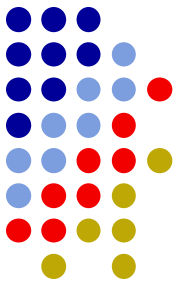


An FD, $A \rightarrow B$, where **A is a set of attributes** and **B is a set of attributes**, is a statement that if two tuples agree on attributes A they must agree on attributes B

If the values of attributes A are known, then the value of attributes B, are known/determined.

For one A value there is ONLY one B value.

FDs are a generalization of keys.



Functional Dependencies: Consider the application domain

Likely **functional dependencies**:

ssn \rightarrow *employee-name*

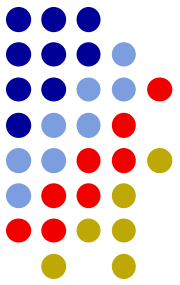
course-number \rightarrow *course-title*

Unlikely **functional dependencies**

course-number ~~\rightarrow~~ *book*

course-number ~~\rightarrow~~ *car-color*

Every key implies a set of FDs



Each key implies a set of functional dependencies (FDs) from the key to the non-key attributes.

Employee (SSN, Name, Salary, Job-code)

A diagram showing three red curved arrows originating from the underlined 'SSN' attribute and pointing to the 'Name', 'Salary', and 'Job-code' attributes respectively, illustrating the functional dependencies implied by the key.

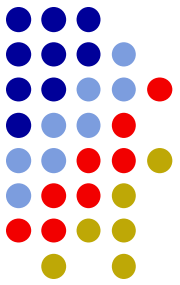
FDs implied by the key:

SSN \rightarrow Name

SSN \rightarrow Salary

SSN \rightarrow Job-code

All of these FDs will be enforced when the DBMS enforces the key.



FDs Determine Keys

If we know these FDs:

SSN → *name*

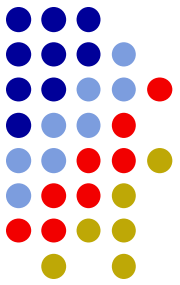
SSN → *hire-date*

SSN → *phone*

then **SSN** is a key for a table with these attributes:

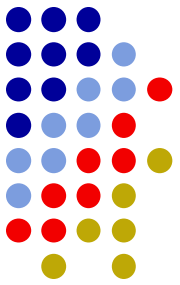
Employee (SSN, name, hire-date, phone)

Quick Exercise



- Work with a partner.
- Identify an application domain.
- List a number of FDs that hold for the domain.
- List a number of FDs that do NOT hold in that domain.
- Define tables for the application.
- List the FDs implied by the key for a table.
- See if you have any problems like we saw with department code and department name (as before).

Keys and FDs (notation)



Given a table R, with a and b (together) as a key for the table, the following FDs are implied by the key.

R (a, b, c, d, e) then

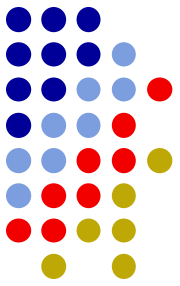
$ab \rightarrow c$

$ab \rightarrow d$

$ab \rightarrow e$

Note we also write this as: $ab \rightarrow cde$

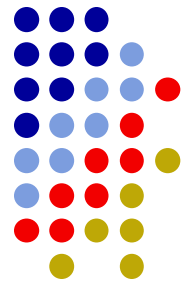
Definition of key/superkey in our book



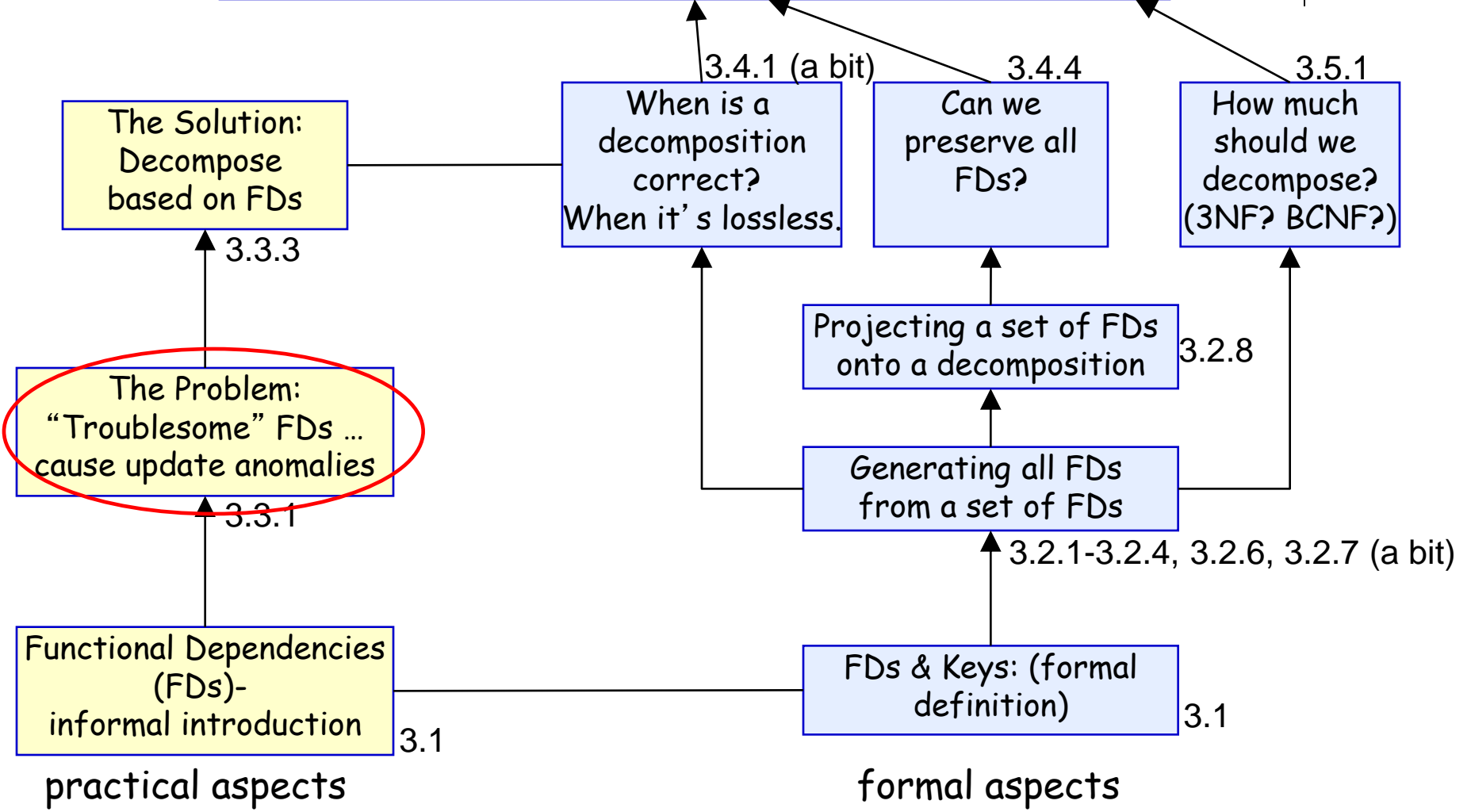
- A set of attributes $\{A_1, \dots, A_n\}$ is a **key** for relation R if and only if:
 - The set of attributes $\{A_1, \dots, A_n\}$ functionally determine all attributes in relation R, and
 - No proper subset of $\{A_1, \dots, A_n\}$ functionally determines all attributes in relation R. (That is, $\{A_1, \dots, A_n\}$ must be a minimal key.)
- A set of attributes $\{A_1, \dots, A_n\}$ is a **superkey** for relation R if and only if $\{A_1, \dots, A_n\}$ contains a key. (superkey = superset of a key)

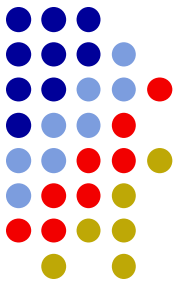
A key is a superkey but a superkey is not necessarily a key.

Normalization Strand Map



Goals for Normalization:
 (1) Lossless (Correct) Required, (2) All tables in BCNF, and
 (3) All FDs Preserved. Sometimes you must choose (2) vs. (3).



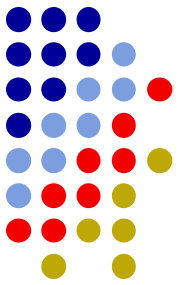


Redundancy (storing info. more than once)

- **Disadvantage:** Any time information is stored more than once, it has the possibility of being **inconsistent** when you update one and not the rest.
 - Phone numbers in your handheld
 - Phone numbers in your cell phone
 - Phone numbers in your address book

If someone changes their phone number, do you remember to change it in every place?

- **Advantage:** **Redundant copies improve retrieval/queries!**



Some Redundancy is Caused by FDs

Consider this table:

EMP(name, SSN, birthdate, address, dnum, dname, dmgr)

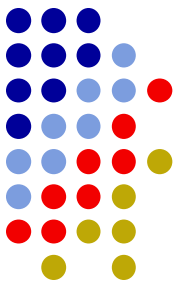
The *dname* and *dmgr* are stored redundantly – whenever there are multiple employees in a department.

This redundancy is caused by what I informally call “troublesome” FDs. The FDs shown in blue are “troublesome” here. A “troublesome” FD is one that is NOT implied by the key for the table.

Note: foreign keys always carry redundant values. We can't get rid of this kind of redundancy. And there can be other redundancy not caused by troublesome FDs.

Redundancy caused by troublesome FD

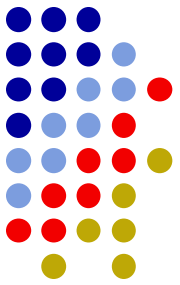
– Sample Data



EMP(name, SSN, birthdate, address, dnum, dname, dmgr)

John	111	June 3	123 St.	D1	sales	222
Sue	222	May 15	455 St.	D1	sales	222
Max	333	Mar. 5	678 St.	D2	research	333
Wei	444	May 2	999 St.	D2	research	333
Tom	555	June 22	888 St.	D2	research	333

We have the department name and manager twice for D1
and three times for D2!



Update anomalies for

EMP(name, SSN, birthdate, address, dnum, dname, dmgr)

Insertion anomalies:

if you insert an employee with a department
then you need to know the descriptive information for
that department.

if you want to insert a department, you can't ... until
there is at least one employee.

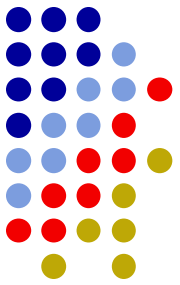
Deletion anomalies: if you delete an employee, is that dept.
gone? Was this the last employee in that dept.?

Modification anomalies: If you want to change *dname*, for
example, you need to change it everywhere! And you
have to find them all first.

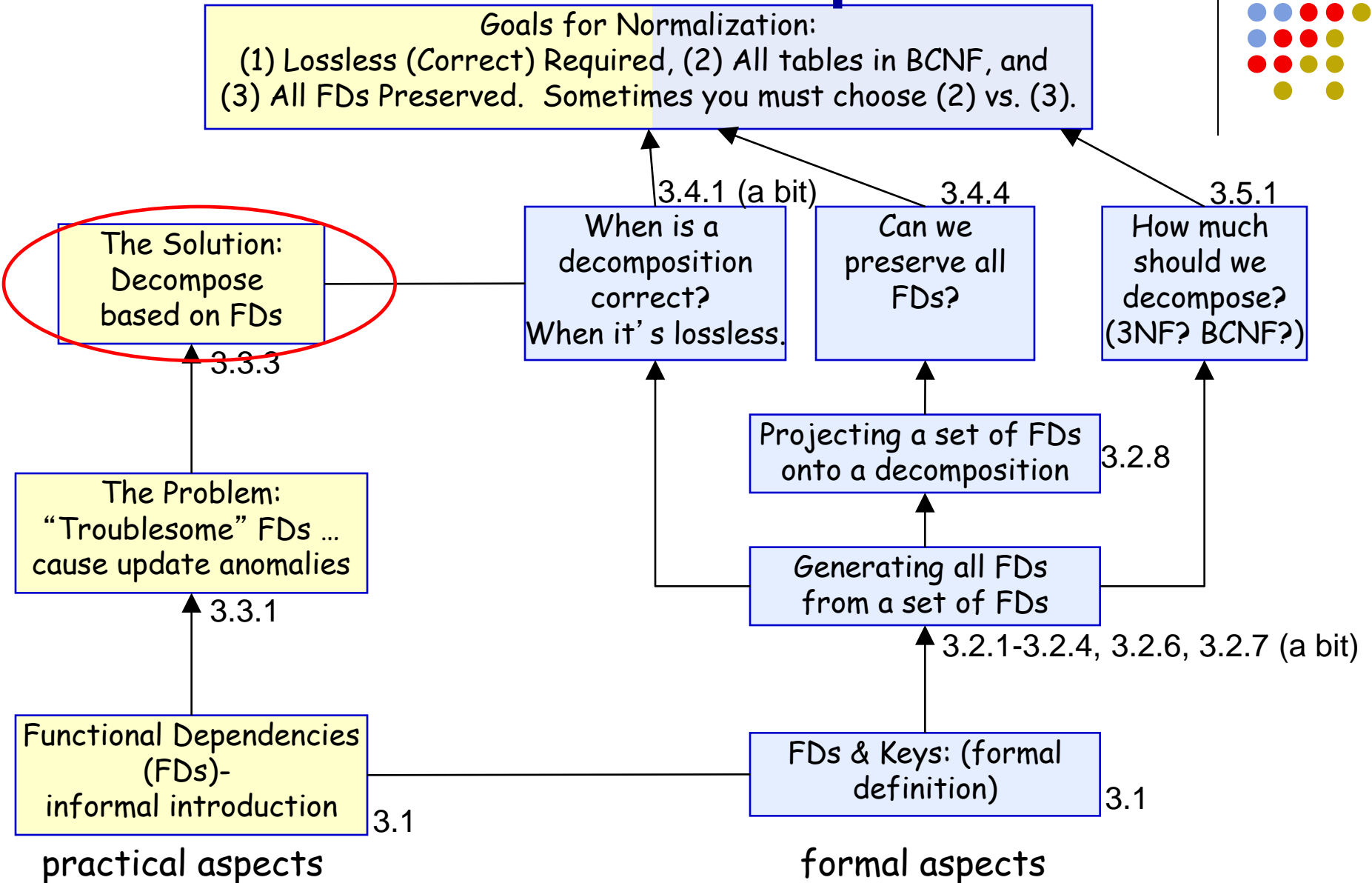
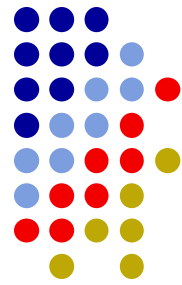
Troublesome FDs cause (redundancy and) update anomalies.

Quick Exercise

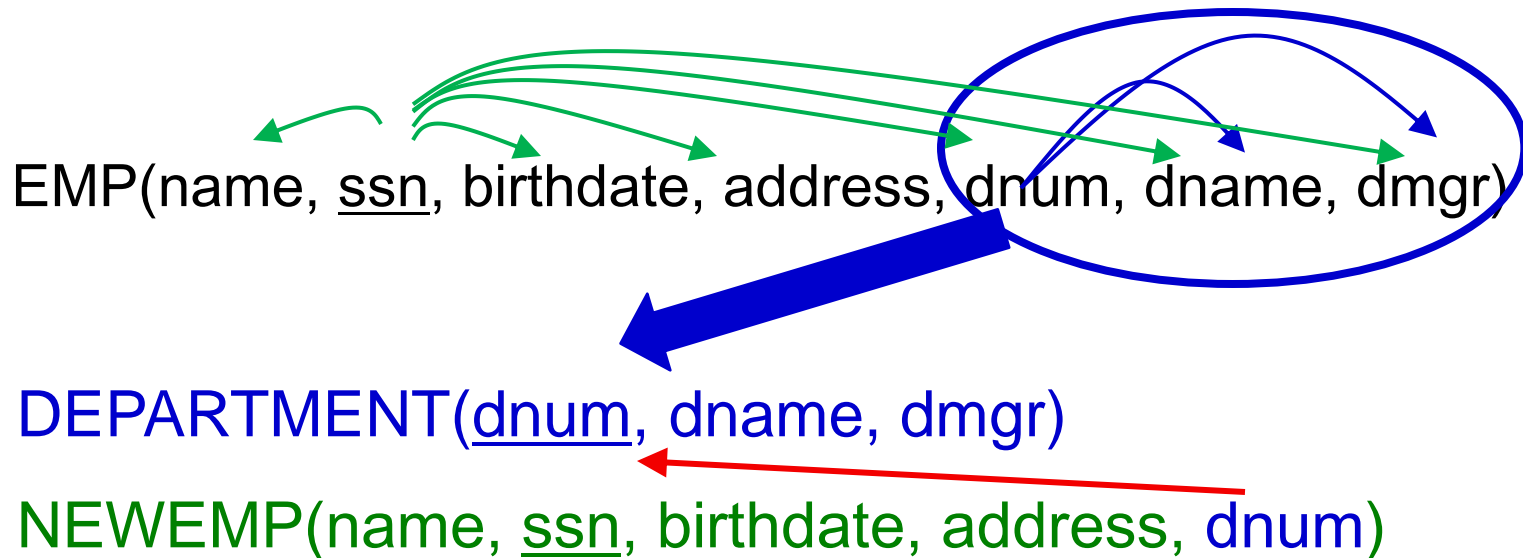
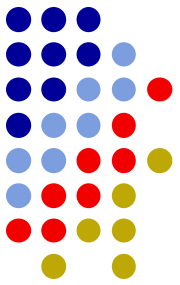
- Define some tables, with key(s), for an application domain that have some troublesome FDs.
- For any troublesome FD, describe the update anomalies that can occur because of the FD.



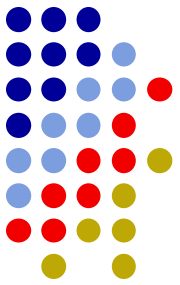
Normalization Strand Map



Example: Decompose relations with troublesome FDs into two



1. Lift the “troublesome” FDs into their own table with LHS as the key. Now they will be enforced.
2. Leave the LHS of the “troublesome” FDs behind. Define a foreign key where Employee.dnum REFERENCES Department.dnum



Are the update anomalies gone?

When this table:

Emp(name, ssn, birthdate, address, dnum, dname, dmgr)

is replaced by these two tables:

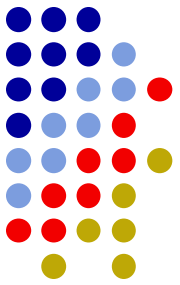
Department(dnum, dname, dmgr)

NewEmp (name, ssn, birthdate, address, dnum)

Let's Check

Department(dnum, dname, dmgr)

NewEmp (name, ssn, birthdate, address, dnum)



Insertion anomalies:

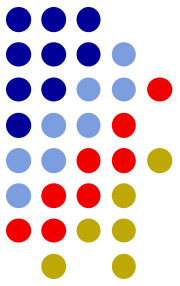
if you insert an employee with a department
then you need to know the descriptive information for
that department. **NO – ONLY THE NUMBER**
if you want to insert a department, you can't ... until
there is at least one employee. **NO PROBLEM**

Deletion anomalies: if you delete an employee, is that dept.
gone? Was this the last employee in that dept.? **NO PROBLEM**

Modification anomalies: If you want to change *dname*, for
example, you need to change it everywhere! And you
have to find them all first. **dname is only stored once!**

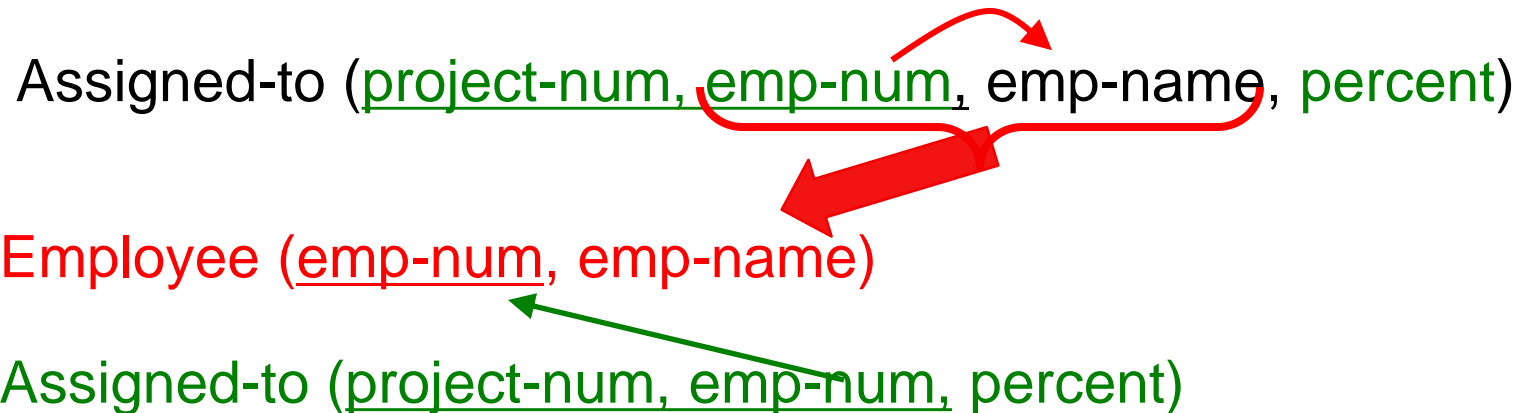
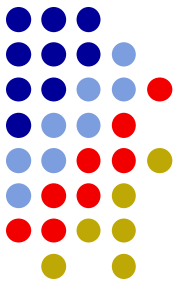
Is there any redundancy? Yes – in the foreign key.

Basic Idea: Normalize (decompose) based on FDs

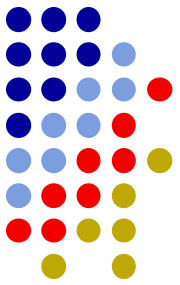


- Identify all the (non-trivial) FDs in an application.
 - Identify FDs that are implied by the keys.
 - Identify FDs that are NOT implied by the keys – the “troublesome” ones.
- Decompose a table with a “troublesome” FD into two or more tables by “lifting” the troublesome FDs into a table of their own. Note: when there are two or more “troublesome” FDs with the same LHS, then they can be lifted, together, into a single table.

Another Decomposition Example



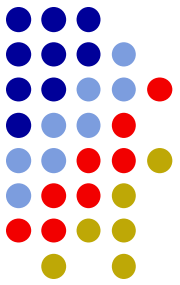
1. Lift the attributes involved in the troublesome FD(s) into a table of their own. Key for new table is left hand side (LHS) of the troublesome FD.
2. Leave the left hand side (LHD) of the FDs behind in the original table.
3. Eliminate *emp-name* from the *Assigned-to* table.



Quick Exercise

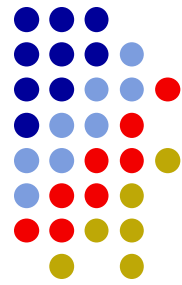
- Define a table with the same pattern for the troublesome FD as the preceding example.
- Explain why the troublesome FD is NOT implied by the key for the table.
- Decompose the table using the “lifting” algorithm just described.
- Check to see whether any of the update anomalies can still occur.

Questions about normalization

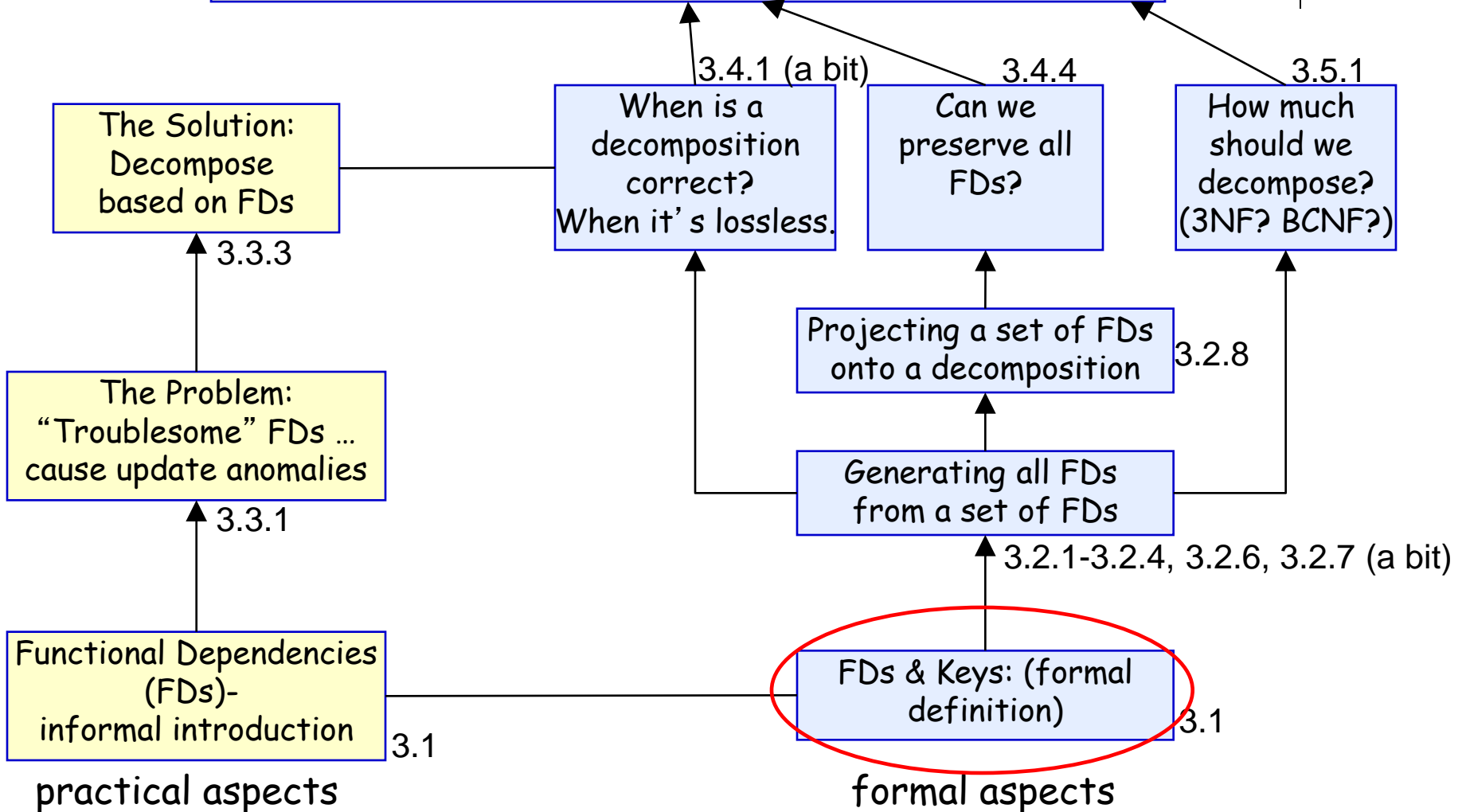


- How do we know which FDs we have?
Talk to domain experts; identify FDs; use them as the starting point for normalization.
- Is the decomposition is correct? Yes, if you use the lifting algorithm based on FDs. (what is correct?)
- How do we know how much to normalize?
How far should we go? Until the troublesome FDs are gone. (can we always do that?)
- How do we know if all of the FDs of interest are being enforced – by using keys for a table? Did we lose any? We need to project the FDs and check.

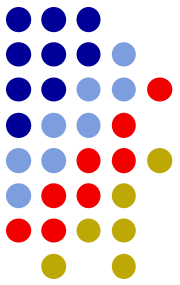
Normalization Strand Map



Goals for Normalization:
 (1) Lossless (Correct) Required, (2) All tables in BCNF, and
 (3) All FDs Preserved. Sometimes you must choose (2) vs. (3).



Definition of a function



Remember the definition of a **function**:

x	$f(x)$
1	2
1	3
2	5
3	5

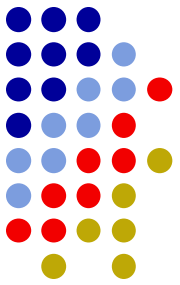
x	$g(x)$
1	2
2	2
3	5

x	$h(x)$
1	10
2	20
3	30

Which of these are functions?

An FD is a **functional** relationship
(that **must** occur in a relation) among attribute values

Answer (concerning functions)



x	f(x)
1	2
1	3
2	5
3	5

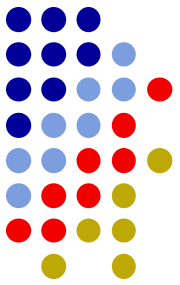
x	g(x)
1	2
2	2
3	5

x	h(x)
1	10
2	20
3	30

f is NOT a function because for an input of “1” there are two answers, “2” and “3”.

g and h are functions.

h is a one-to-one (injective) function.



Example of an FD – a function

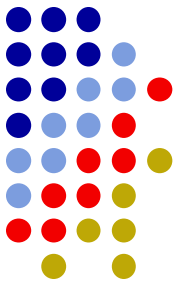
Employee (ssn, name, phone, salary)

Since *ssn* → *name* is an FD

we know that there is only one name for an ssn.

Thus we know that *ssn and name* are in a
functional relationship!

Note: we still need to store this relation on disk;
there is no way to compute name from ssn.



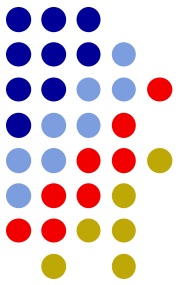
Another Example

Employee (ssn, name, phone, dept, dept-mgr)

dept → *dept-mgr*

We know that there is only one dept-mgr for a dept.

We know that *dept* → *dept-mgr* is a **function!**



Trivial FD

We have a trivial FD whenever the attributes on the right side of an FD are a subset of the attributes on the left side of the FD:

$$A \rightarrow A$$

$$AB \rightarrow A$$

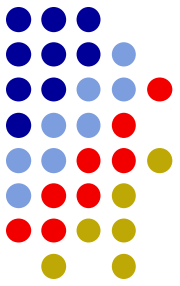
$$ABCD \rightarrow BCD$$

A trivial FD represents an function: $f(X) = X$

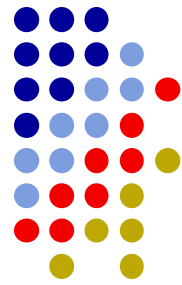
It is definitely a function ... and definitely an FD. But it's not "troublesome" and won't help us decompose a table. There are LOTS of trivial FDs.

Quick Exercise

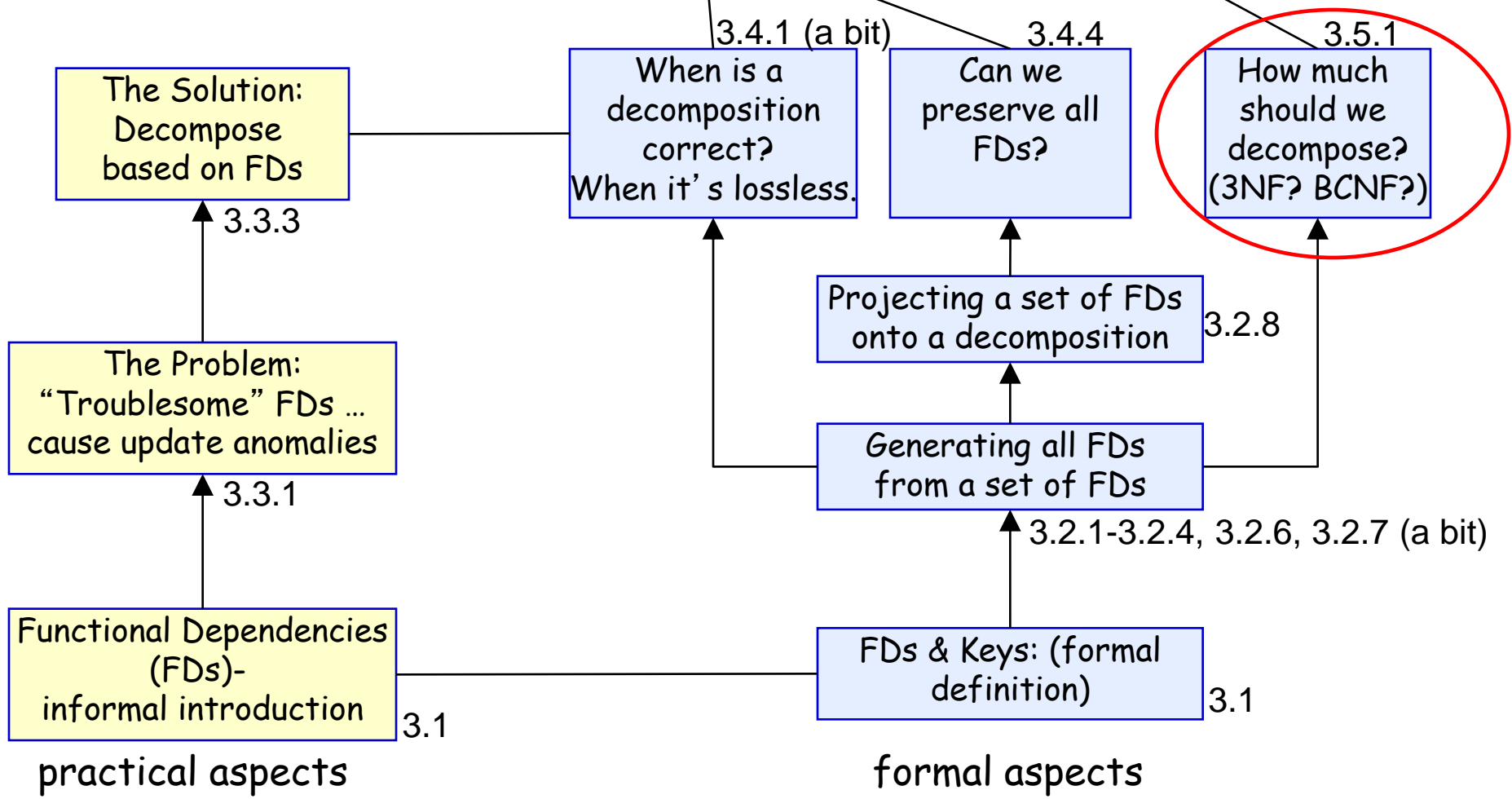
- For a table that you have defined, write down some trivial FDs.
- Write down some non-trivial FDs.
- Write down the definition of a trivial and a non-trivial FD.



Normalization Strand Map

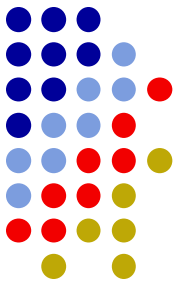


Goals for Normalization:
 (1) Lossless (Correct) Required, (2) All tables in BCNF, and
 (3) All FDs Preserved. Sometimes you must choose (2) vs. (3).



Informal Definitions

Normal Forms Based on FDs



1NF - all attribute values (domain values) are atomic
(part of the definition of the relational model)

2NF - all non-key attributes must depend on the **whole** key (**no partial dependencies**)

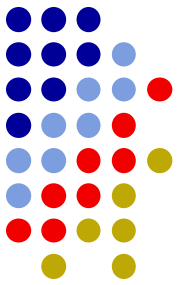
r (A B C D E) **B → C violates 2NF**

3NF – table is in 2NF and all non-key attributes must depend on **only** a key (**no transitive dependencies**)

r (A B C D E)

BCNF - every **determinant** (LHS of an FD) is a **key** for the table
(All FDs are implied by the keys)

r (A B C D E)



Examples of Violations of 2NF, 3NF, BCNF


2NF - all **non-key** attributes must depend on a whole key

Assigned-to (A-project, A-emp, emp-name, percent)



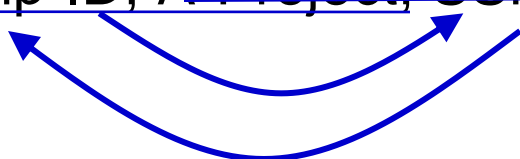
3NF – 2NF and all **non-key** attributes must depend on only a key

Employee (SSN, name, address, project, p-title)



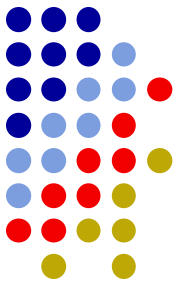
BCNF - every determinant (LHS of an FD) is a superkey for this table (all FDs are implied by the keys)

Assigned-to (Emp-ID, A-Project, SSN, percent)

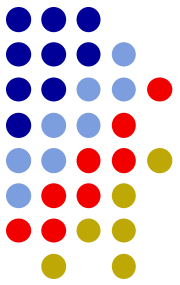


Notes from previous slide

- Every violation of one of the normal forms is based on a “troublesome” FD.
- The solution (to get rid of the troublesome FDs) is to use the decomposition algorithm that you’ve already seen.



Fix violations of 2NF by lifting “troublesome” FDs



(Example repeated from earlier slide)

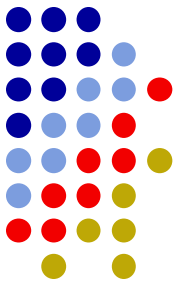
Assigned-to (A-project, A-emp, emp-name, percent)

Employee (A-emp, emp-name)

1. Lift the troublesome FD(s) into a table of their own.
Key for new table is left hand side of the troublesome FD.
2. Leave the left side of the FD behind in the original table;
eliminate the RHS of the FD. Introduce a foreign key from
the LHS here to the LHS in the new table.

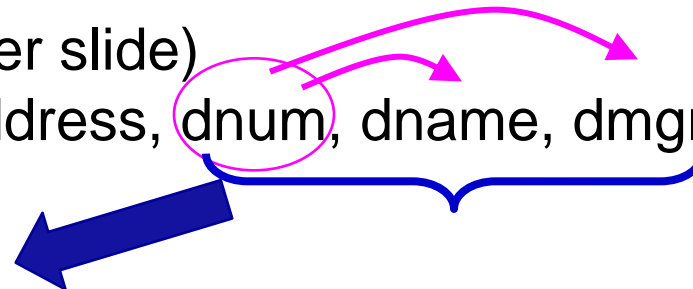
Assigned-to (A-project, A-emp, percent)

Fix violations of 3NF by “lifting” troublesome FDs



(Example repeated from earlier slide)

Emp(name, ssn, birthdate, address, dnum, dname, dmgr)

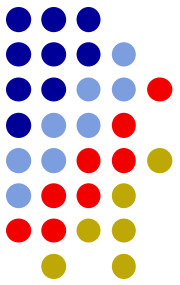


Department(dnum, dname, dmgr)

1. Lift the troublesome FD(s) into a table of their own.
Key for new table is left hand side of the troublesome FD.
2. Leave the left side of the FD behind in the original table.
Eliminate *dname* and *dmgr* from the *NewEmp* table.
Introduce a foreign key from *dnum* to *Department.dnum*

NewEmp (name, ssn, birthdate, address, dnum)

Fix violations of BCNF by lifting “troublesome” FDs



BCNF - every determinant is a key (all FDs implied by the keys)

Assigned-to (A-emp, A-project, A-ssn, percent)

Take the troublesome
FD and put it into
a table of it's own.

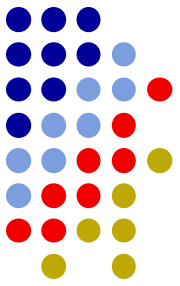
Employee (A-emp, A-ssn)

Then leave the left side of the troublesome FD in the original
table:

Assigned-to (A-emp, A-project, percent)

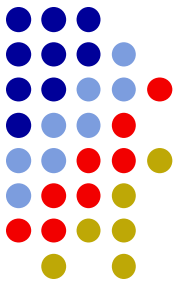
Formal definition of BCNF

(in the textbook)



- For a table R , for every FD $X \rightarrow A$ that occurs among attributes of R then either:
 - A is an element of X ($X \rightarrow A$ is trivial) or
 - X is a superkey of R .
- Basically, a relation is in BCNF if every non-trivial FD is implied by the keys for the relation.
- 3NF is weaker than BCNF. For a table R , for every FD $X \rightarrow A$ that occurs among attributes of R then either:
 - A is an element of X ($X \rightarrow A$ is trivial) or
 - X is a superkey of R or
 - the attributes in X that are not in A are part of some key for the table.

One Goal of Normalization - BCNF



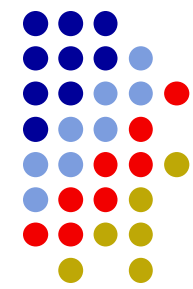
We would like to decompose until all tables are in BCNF.

Then, there are no remaining redundancies (and update anomalies) caused by FDs.

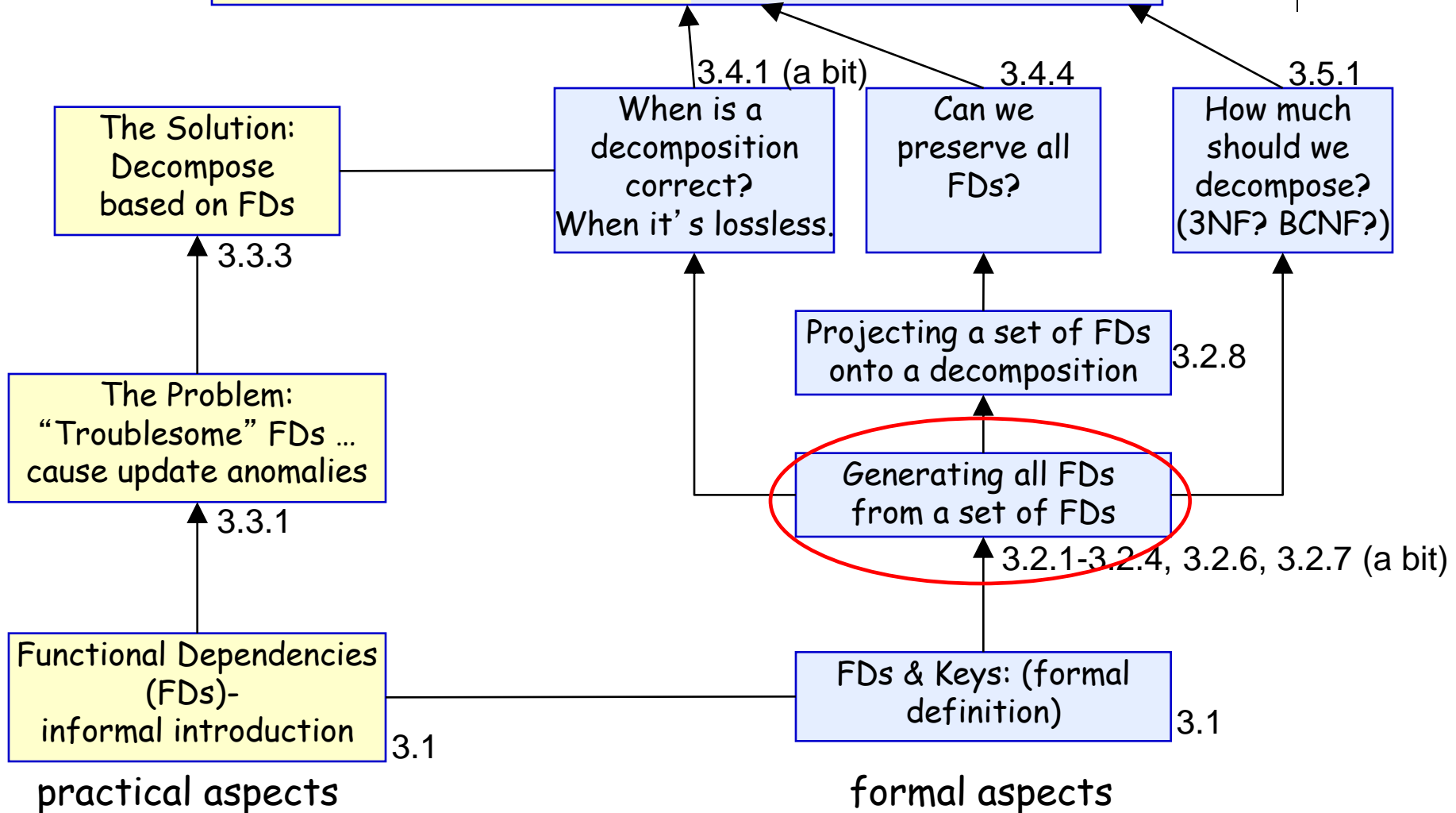
Algorithm:

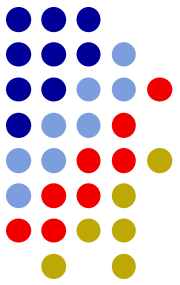
1. Find a non-trivial FD where the LHS is not a superkey for the relation.
2. Decompose based on that FD (by lifting it out into another relation and leaving the LHS behind in the original relation).
3. Continue until no such FD can be found.

Normalization Strand Map



Goals for Normalization:
(1) Lossless (Correct) Required, (2) All tables in BCNF, and
(3) All FDs Preserved. Sometimes you must choose (2) vs. (3).





Use rules of inference for FDs

- Splitting/combining rule:

- $a \rightarrow b$
- $a \rightarrow c$
- $a \rightarrow d$
- $a \rightarrow e$

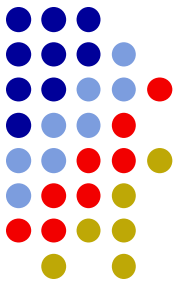
Is equivalent to:

- $a \rightarrow bcde$

- Trivial dependency rule:

- Take out attributes from the right-hand side that also appear in the left-hand side
- $abc \rightarrow cd$ is equivalent to:
- $abc \rightarrow d$

- Transitive rule: $a \rightarrow b, b \rightarrow c$, then $a \rightarrow c$.

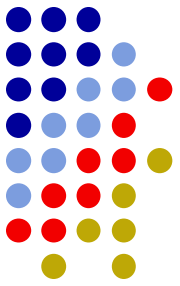


Quick Exercise

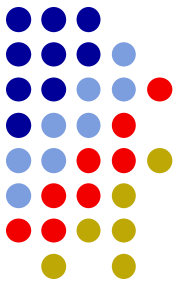
- Define a table in an application domain where you have two FDs – where the transitive property applies.
- Prepare some sample data to convince yourself that the FDs that results from the use of the transitive rule really is an FD.

Transitive rule: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Closure of a set of attributes (e.g., A_1)



- Start with a set of attributes $X = \{A_1, \dots, A_n\}$ and a set of FDs S
- First, split all FDs so that the RHS consists of just one attribute.
- Choose the set of attributes to close.
Let $X = \{A_1\}$ for example.
- Search for an FD in S $B_1 \dots B_m \rightarrow C$ where $\{B_1, \dots, B_m\}$ is a subset of X and C is not in X . Add C to X . Repeat until you can't find any more such FDs.
- Finally, $\{A_1\}$ determines every attribute in X (and no other attributes – based on S).



Quick Exercise

- Consider this table:
 $X(\text{sid}, \text{sname}, \text{age}, \text{tid}, \text{tname})$
with these FDs:

$\text{sid} \rightarrow \text{sname}$

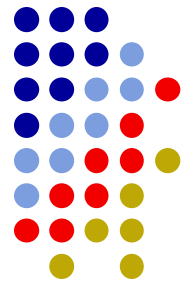
$\text{sid} \rightarrow \text{age}$

$\text{sid} \rightarrow \text{tid}$

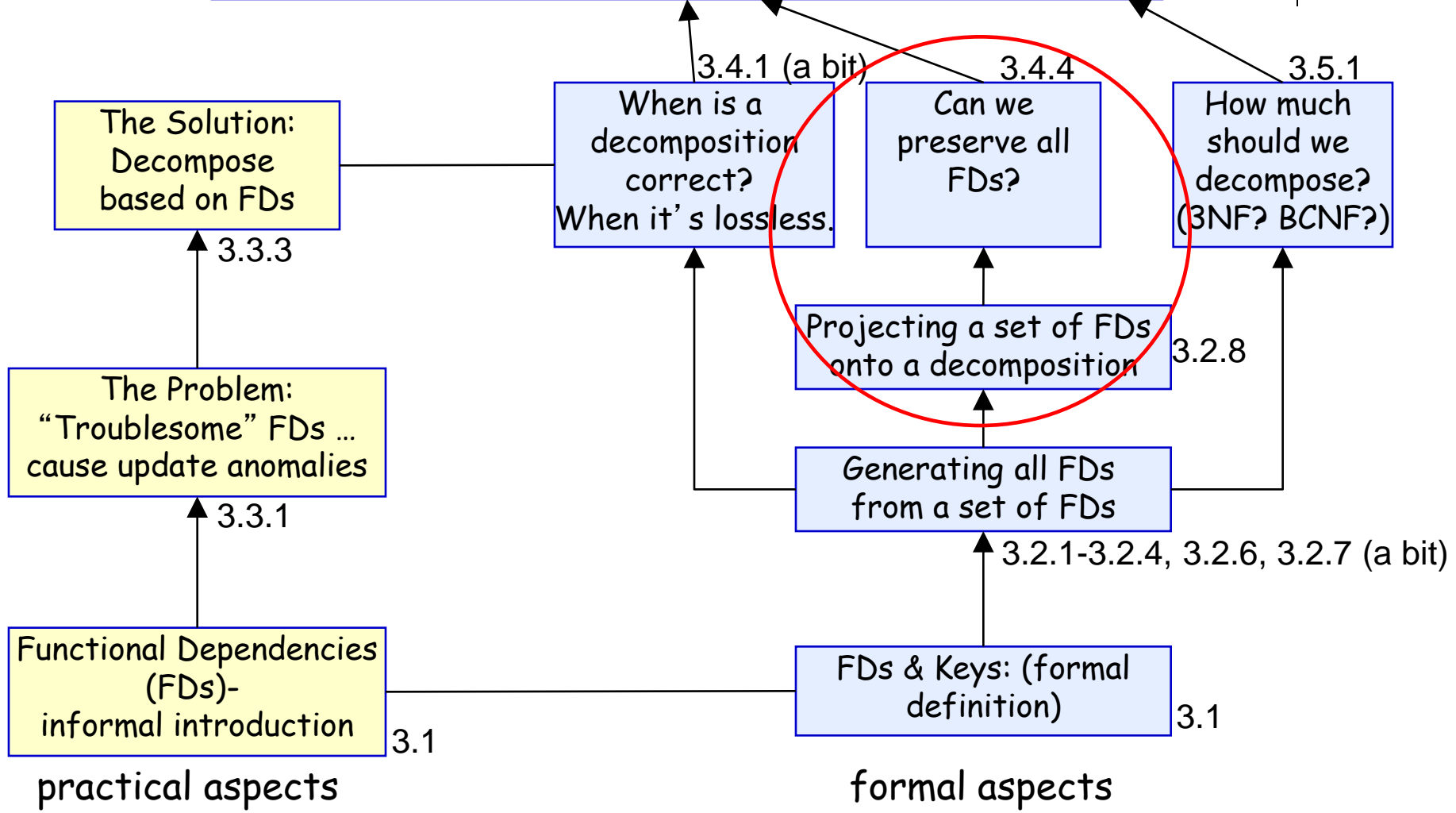
$\text{tid} \rightarrow \text{tname}$

- Compute the closure of sid and of tid . Then decide what the key is for this table.

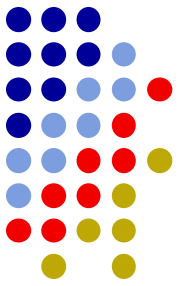
Normalization Strand Map



Goals for Normalization:
 (1) Lossless (Correct) Required, (2) All tables in BCNF, and
 (3) All FDs Preserved. Sometimes you must choose (2) vs. (3).



Losing an FD

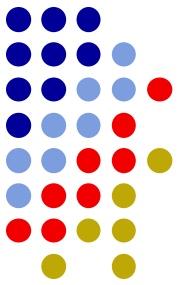


If I have an FD $\text{dept} \rightarrow \text{mgr}$

and if dept ends up in one table and
 mgr ends up in another table

Then this FD will NOT be enforced. (The only way it will be enforced if the attributes are in the same table and dept is the key for the table.)

When we decompose, we need to project a set of FDs onto the tables that we end up with, after decomposition. Then ... find all FDs implied by the ones that are left (using the closure algorithm).



Example: do we lose an FD?

Employee (SSN, name, phone, dept, dept-name)

Original FDs F:

SSN → name SSN → phone SSN → dept
SSN → dept-name dept → dept-name

Employee (SSN, name, phone, dept)

Department (dept, dname)

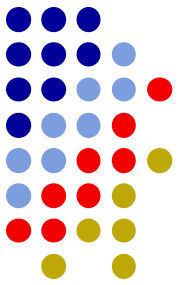
Resulting FDs G:

SSN → name SSN → phone SSN → dept
dept → dept-name

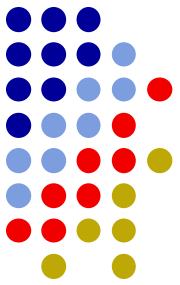
What about **SSN → dept-name**? Is it lost? Can there be two department names for one SSN?

NO! It's not lost. One SSN has only one dept. And one dept has only one dept-name. So SSN has only one dept-name.

Projecting a set of FDs onto a decomposition



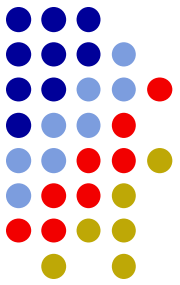
- Given a relation R and a relation $R_1 = \pi_L R$ and a set of FDs, S , that hold in R .
- Compute the closure of all combinations of attributes in R_1 based on the original set of FDs.
- This computes the set of all FDs (from the closure of the original set) that could possibly be preserved in R_1 , the projection.
- Whenever you decompose R into two relations, R_1 and R_2 , compute the projection of S onto the R_1 and R_2 . and take the union of those two sets of FDs. Those are the only ones remaining; those are the only ones in the projection.



Quick Exercise

- Define a table that has one or more troublesome FDs.
- Write down the set of FDs: the ones implied by the key(s) for the table and the troublesome ones.
- Decompose it (normalize it) using the decomposition/lifting algorithm.
- Then ... project the FDs onto the tables that you end up with. Check to see whether all of the original FDs are still there. (You may need to use the rules of inference; close a set of attributes.)

Definition of Dependency Preserving



Suppose F is the original set of FDs.

Compute F^+ .

G is set of FDs from F^+ that are present in individual relations in G . (We project F^+ to relations in the resulting, decomposed schema.)

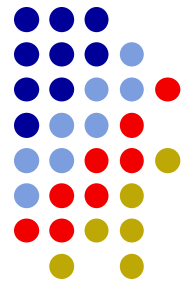
Compute G^+ .

If $F^+ = G^+$

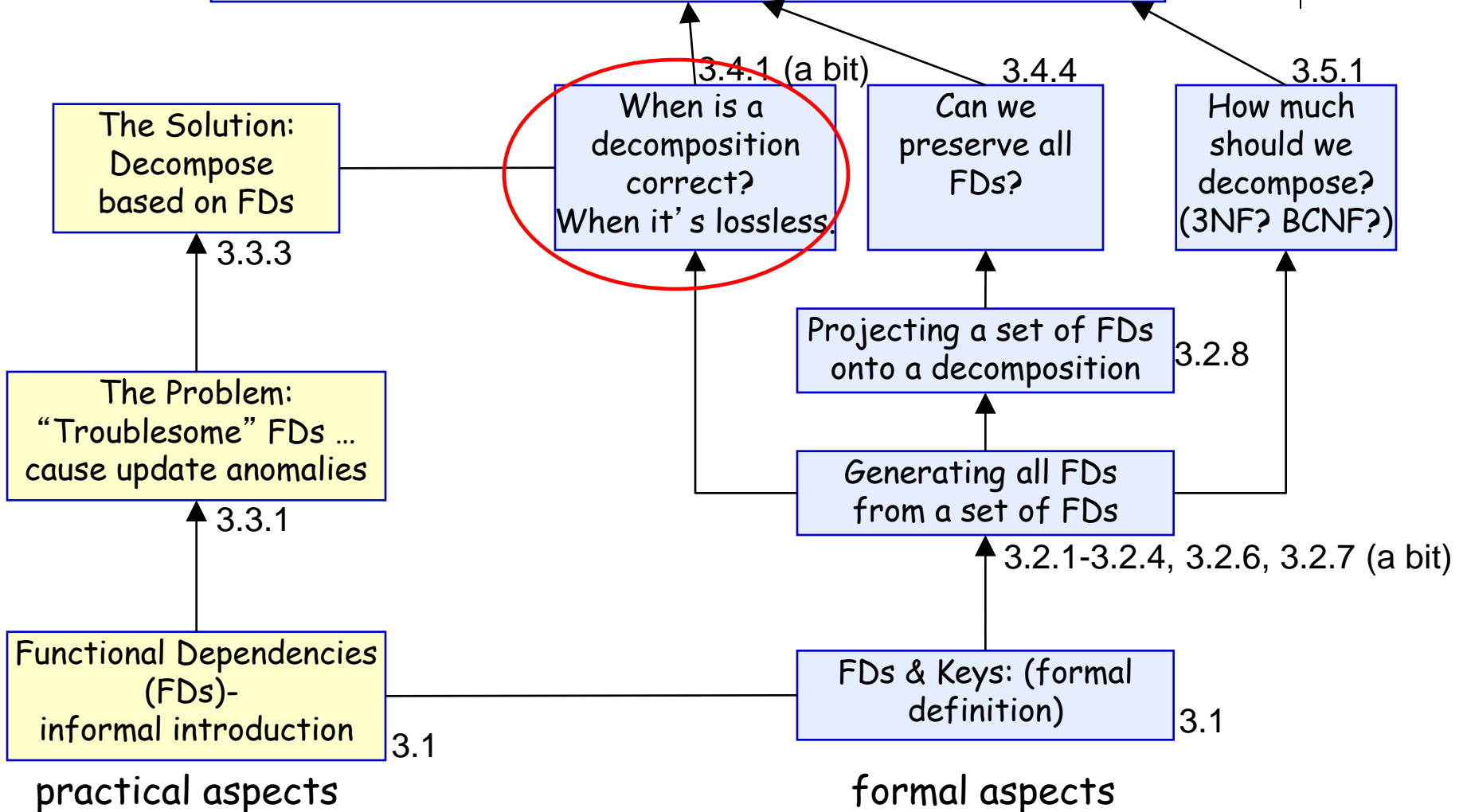
then the decomposition is **dependency preserving**

For a complex design, you may want to implement one of the known algorithms for computing F^+ and G^+ .

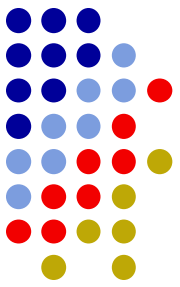
Normalization Strand Map



Goals for Normalization:
 (1) Lossless (Correct) Required, (2) All tables in BCNF, and
 (3) All FDs Preserved. Sometimes you must choose (2) vs. (3).



The “lifting” algorithm is correct; we can join the tables back together.



When

$\text{Emp}(\text{name}, \underline{\text{SSN}}, \text{birthdate}, \text{address}, \text{dnum}, \text{dname}, \text{dmgr})$

is replaced by these two tables:

$\text{Department}(\underline{\text{dnum}}, \text{dname}, \text{dmgr})$

$\text{NewEmp}(\text{name}, \underline{\text{SSN}}, \text{birthdate}, \text{address}, \text{dnum})$

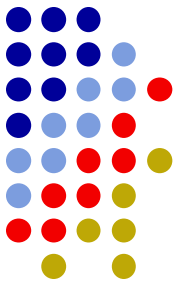
We use the project operator to decompose

$\text{Department} = \pi_{\text{dnum}, \text{dname}, \text{dmgr}} \text{Emp}$

$\text{NewEmp} = \pi_{\text{name}, \text{SSN}, \text{birthdate}, \text{address}, \text{dnum}} \text{Emp}$

And we use the join operator to put the pieces together

$\text{Emp} = \text{Department} \bowtie_{\text{D.SSN}=\text{NewEmp.SSN}} \text{NewEmp}$



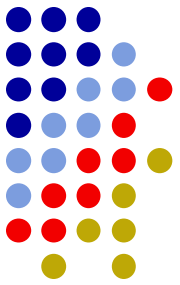
What is a lossless (and a lossy) decomposition?

We want to make sure that we haven't thrown away any information from the original schema.

When table R is decomposed into tables R1 and R2 then the decomposition is **lossless (correct)** if:

$(R1 \bowtie R2)$ is identical to R **natural join**

If it is a **lossy** decomposition, then $R1 \bowtie R2$ gives you **TOO MANY** tuples.



Example: a lossy decomposition

original

Employee(SS-number, name, p-num, p-title)

1	smith	p1	accounting
2	jones	p1	accounting
3	smith	p2	billing

decomposition:

Employee (SS-number, name)

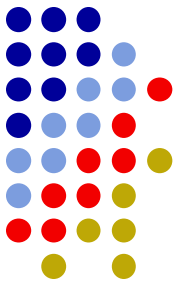
1	smith
2	jones
3	smith

Project (p-num, p-title, name)

p1	account	smith
p1	account	jones
p2	billing	smith

now with natural join: you get at least **one extra tuple!!!**

1 smith p2 billing



Example: Test for a Lossless Decomposition

Consider a table:

R (a, b, c, d, e) with a troublesome FD $d \rightarrow e$.

Decompose it into two tables:

R1(a, b, c, d)

R2(d, e)

As long as

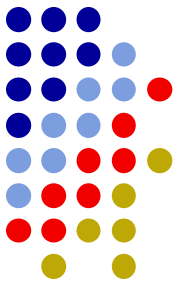
the attributes in common are a key for (at least) one of the relations, R_1 or R_2

then we know that the decomposition is lossless!

For this example d is the attribute in common.

And d is a key for R2, the second table.

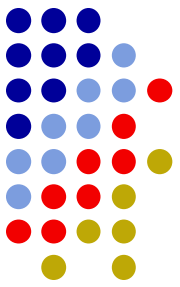
Is the Decomposition Algorithm Lossless?



1. Lift the “troublesome” FD(s) (all the FDs with the same LHS) into a table of their own. Key for new table is left hand side of the troublesome FD(s).
2. Leave the left side of the FD behind in the original table.
3. Eliminate the RHS attributes from the original table.

Yes, we are guaranteed that the decomposition is lossless. The attribute in common is definitely a key for the new “lifted” table.

Example of a Lossy Decomposition (revisited)



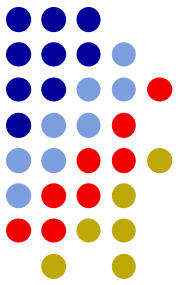
Employee(SS-number, name, project, p-title)

decomposition: Employee (SS-number, name)
Project (p-num, p-title, name)

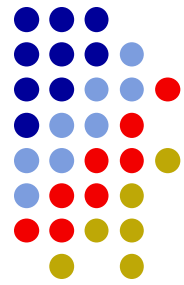
Notice that the common attribute, **name**, is not a key for either of these tables.

Quick Exercise

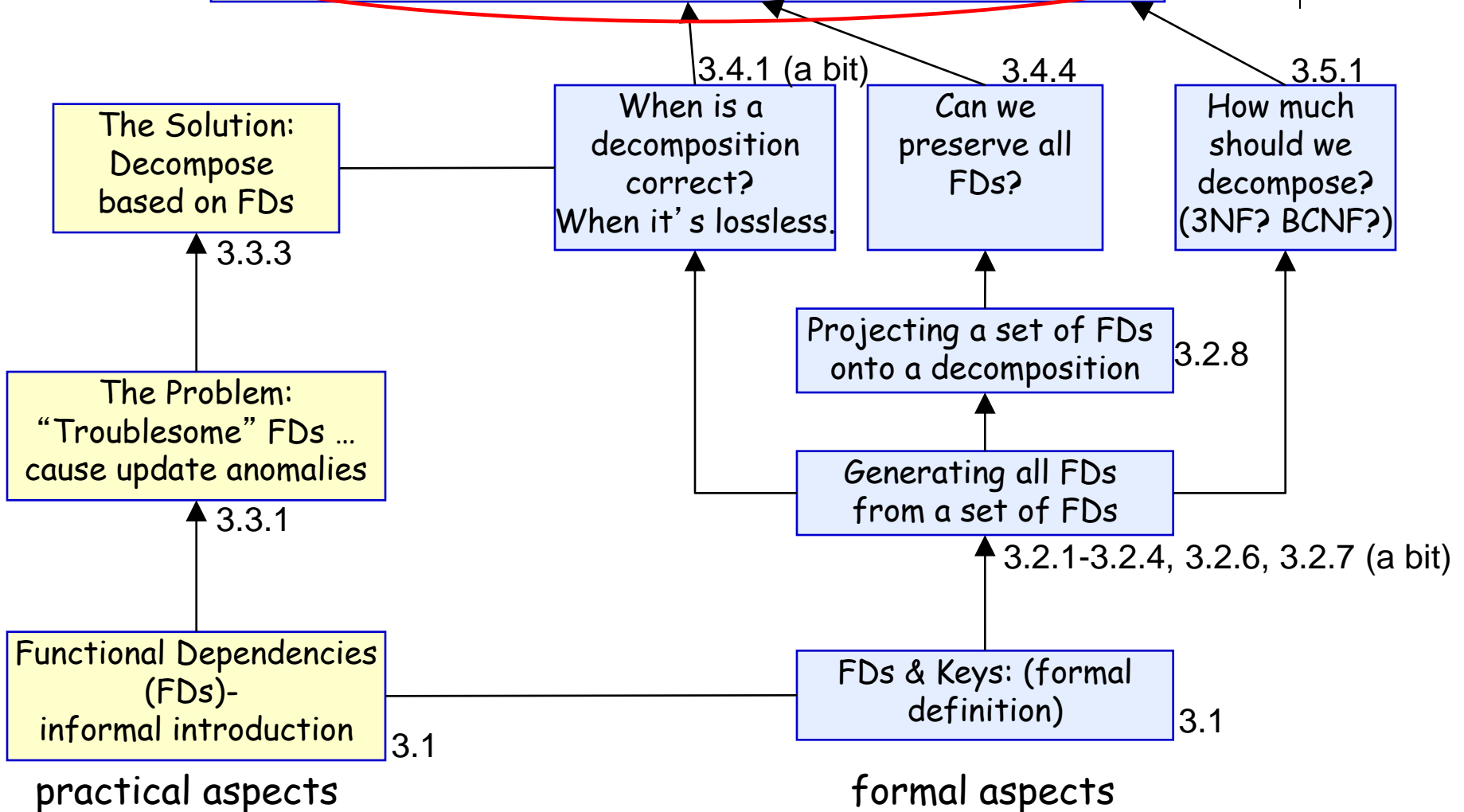
- Define a table (that needs to be normalized).
- Decompose it in a lossy way.
- Use some sample data to show that when you join the resulting tables back together, you get MORE data that you had originally.

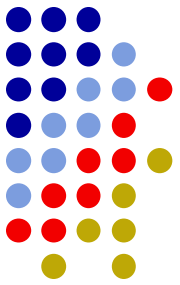


Normalization Strand Map



Goals for Normalization:
 (1) Lossless (Correct) Required, (2) All tables in BCNF, and
 (3) All FDs Preserved. Sometimes you must choose (2) vs. (3).





Three Goals for Normalization

lossless decomposition

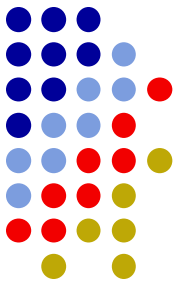
don't throw any information away

be able to reconstruct the original relation

dependency preservation

all of the original, non-trivial FDs can be derived from FDs implied by the keys of resulting tables

Boyce-Codd normal form (BCNF) - no redundancy beyond foreign keys; all FDs implied by keys



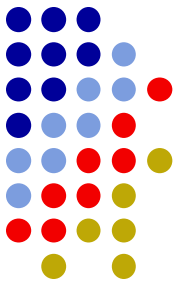
It is not always possible to have BCNF AND dependency preservation

Required! → **lossless** decomposition

Desirable
but not
always
possible
to have both

→ dependency preservation

→ **Boyce-Codd normal form (BCNF)**



Counterexample

(a table that can't be decomposed into BCNF with dependency preservation)

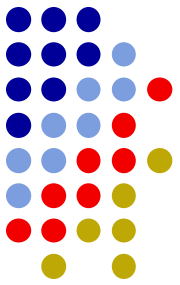
Original table – a table that holds US addresses

addr(number street city state zip)

The original FDs are:

number street city state -> zip

zip -> state



Counterexample (cont.)

Based on the FDs:

number street city state \rightarrow *zip*

zip \rightarrow *state*

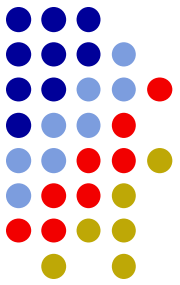
There are two keys for this table

addr(number street city state zip)

The diagram illustrates two keys for the table. The first key is 'number street city state', indicated by a blue underline. The second key is 'number street city state zip', indicated by a red underline. A red arrow points from 'zip' back to 'state', showing that 'zip' is a superkey for 'state'.

Since all attributes are key attributes, this table is automatically in 3NF and 2NF.

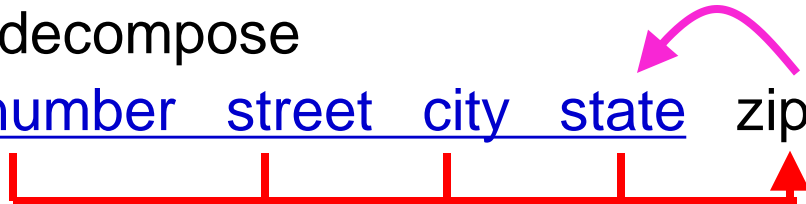
But *zip* \rightarrow *state* violates BCNF



Counterexample (cont.)

Let's decompose

addr(number street city state zip)



using this "troublesome" FD:

zip -> state

Addr2 (number, street, city, zip)

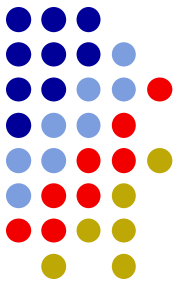
Zip-state (zip, state)

We've lost the FD *number street city state -> zip*

If we put this table back in the design, we are back where we started. And we violate BCNF.

Algorithm for lossless join decomposition into BCNF relations

(not necessarily dependency preserving)



1. set $D := \{ R \}$ (the current set of relations)

2. while there is a relation in R that is not in BCNF
begin

choose a relation Q that is not in BCNF

find an FD $X \rightarrow Y$ in Q that violates BCNF

replace Q in D by two relations: $(Q - Y)$ and $(X \cup Y)$

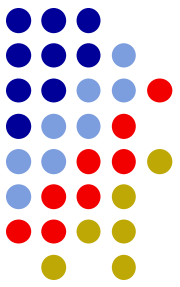
end;

Lifting “troublesome” FD

Finding a “troublesome” FD

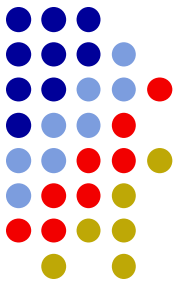
Algorithm for dep. preserving, lossless join decomposition into ~~BCNF~~ 3NF relations

(another example of available algorithms)



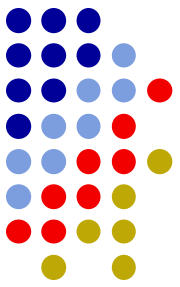
1. set $D := \{ R \}$ (the current set of relations)
2. while there is a relation in R that is not in BCNF
begin
 choose a relation Q that is not in BCNF
 find an FD $X \rightarrow Y$ in Q that violates BCNF
 replace Q in D by two relations: $(Q - Y)$ and $(X \cup Y)$
end;
3. identify dependencies that are not preserved ($X \rightarrow A$).
 add XA as a table to the set D

dep preserving
Same as before



A few comments

Sometimes redundancy has NOTHING to do with FDs



Suppose we have two tables for employee information.

Employee(ssn, name, salary, birthdate)

Employee2(ssn, name, home-address)

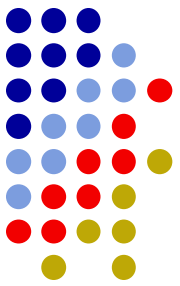
And, we put the name in both tables, for convenience.

Name is stored redundantly.

And, it is possible for the two names to be inconsistent, if you change it in one place but not in another.

This redundancy is NOT caused by a troublesome FD

Sometimes redundancy has NOTHING to do with FDs



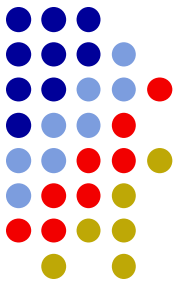
Foreign keys are necessarily ... by definition ... introducing redundancy.

Student(id, name, major, advisor-num)

Faculty(id, name, rank)

The faculty.id value is repeated in the Student.advisor-num attribute for every student that has this faculty member as an advisor.

Null Values are Useful



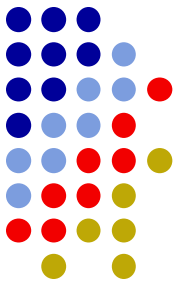
Makes the DB more flexible; makes it simpler to insert data, for example

For example:

Employee(ssn, name, DOB, spouse)

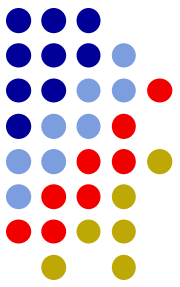
We might not know the DOB (when data is entered) or this employee might not have a spouse.

Null values cause problems



- may waste space
- may have different meanings:
 - attribute *does not apply* to this tuple
 - attribute value is *unknown*
 - value is *known but absent* (not yet recorded)
- may make queries harder to write
 - need to use outer joins (rather than joins)
 - may make aggregates harder to understand

Null Values Cause Problems for Aggregate Operators



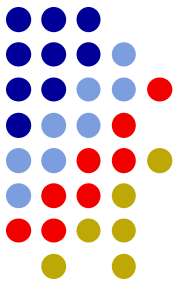
Employee (ssn, name, salary)

```
SELECT AVG(salary) FROM Employee;
```

```
SELECT SUM(salary) INTO salsum FROM Employee;  
SELECT COUNT(*) INTO total FROM Employee;
```

Salsum/total might be different from first query answer. How could that happen?

Decomposition Reduces the Use of Null Values



Use two tables:

Employee (ssn, name, DOB)

Employee-extra (ssn, spouse)

Rather than:

Employee(ssn, name, DOB, spouse)

Generally, it is better to reduce the use of null values, if you can. The first design, above, doesn't require the use of null values for spouse.