Overview - detailed

- DB design and normalization
  - pitfalls of bad design
  - decomposition
  - normal forms

Goal

- Design ‘good’ tables
  - sub-goal #1: define what ‘good’ means
  - sub-goal #2: fix ‘bad’ tables
- in short: “we want tables where the attributes depend on the primary key, on the whole key, and nothing but the key”
- Let’s see why, and how:

Pitfalls

takes 1 (ssn, c-id, grade, name, address)

<table>
<thead>
<tr>
<th>Ssn</th>
<th>c-id</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>413</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
</tbody>
</table>
Pitfalls

‘Bad’ - why? because: ssn->address, name

<table>
<thead>
<tr>
<th>Ssn</th>
<th>c-id</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>413</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>415</td>
<td>B</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>211</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
</tbody>
</table>

Pitfalls?

• Redundancy
  – ??
  – ??

Pitfalls

• Redundancy
  – space
  – (inconsistencies)
  – insertion/deletion anomalies:

Pitfalls

• insertion anomaly:
  – “jones” registers, but takes no class - no place to store his address!

<table>
<thead>
<tr>
<th>Ssn</th>
<th>c-id</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>413</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>234</td>
<td>null</td>
<td>null</td>
<td>jones</td>
<td>Forbes</td>
</tr>
</tbody>
</table>
Pitfalls

• deletion anomaly:
  – delete the last record of ‘smith’ (we lose his address!)

<table>
<thead>
<tr>
<th>Ssn</th>
<th>C-id</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>413</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>415</td>
<td>B</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>211</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
</tbody>
</table>

Solution: decomposition

• split offending table in two (or more), eg.:

<table>
<thead>
<tr>
<th>Ssn</th>
<th>C-id</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>413</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>415</td>
<td>B</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>211</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
</tbody>
</table>

Overview - detailed

• DB design and normalization
  – pitfalls of bad design
  – decomposition
    • lossless join decomp.
    • dependency preserving
  – normal forms

Decompositions

There are ‘bad’ decompositions. Good ones are:
• lossless and MUST HAVE
• dependency preserving Nice to have
Decompositions - lossy:

R1(ssn, grade, name, address)  R2(c-id, grade)

<table>
<thead>
<tr>
<th>Ssn</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>B</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>234</td>
<td>A</td>
<td>Jones</td>
<td>Forbes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C-id</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>413</td>
<td>A</td>
</tr>
<tr>
<td>211</td>
<td>A</td>
</tr>
</tbody>
</table>

- ssn->name, address
- ssn, c-id -> grade

Decompositions - lossy:
can not recover original table with a join!

<table>
<thead>
<tr>
<th>Ssn</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>B</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>234</td>
<td>A</td>
<td>Jones</td>
<td>Forbes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C-id</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>413</td>
<td>A</td>
</tr>
<tr>
<td>211</td>
<td>A</td>
</tr>
</tbody>
</table>

- ssn->name, address
- ssn, c-id -> grade

Decompositions - overview

There are ‘bad’ decompositions. Good ones are:

- lossless and MUST HAVE
- dependency preserving Nice to have

Decompositions

example of non-dependency preserving

<table>
<thead>
<tr>
<th>S#</th>
<th>address</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>London</td>
<td>E</td>
</tr>
<tr>
<td>125</td>
<td>Paris</td>
<td>E</td>
</tr>
<tr>
<td>234</td>
<td>Pitts.</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S#</th>
<th>address</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>London</td>
<td>E</td>
</tr>
<tr>
<td>125</td>
<td>Paris</td>
<td>E</td>
</tr>
<tr>
<td>234</td>
<td>Pitts.</td>
<td>A</td>
</tr>
</tbody>
</table>

- S# -> address, status
- address -> status
Decompositions - overview

There are ‘bad’ decompositions. Good ones are:

• #1) lossless and MUST HAVE
• #2) dependency preserving Nice to have

How to automatically determine #1 and #2?

Decompositions - lossless

Definition:
consider schema R, with FD ‘F’. R1, R2 is a lossless join decomposition of R if we
always have: r1⋈r2 = r

An easier criterion?

Decomposition - lossless

Theorem: lossless join decomposition if the joining attribute is a superkey in at least one of the new tables

Formally:
R1 ∩ R2 → R1 or
R1 ∩ R2 → R2

Decomposition - lossless

example:

<table>
<thead>
<tr>
<th>Ssn</th>
<th>c-id</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>413</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>415</td>
<td>B</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>234</td>
<td>211</td>
<td>A</td>
<td>jones</td>
<td>Forbes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ssn</th>
<th>c-id</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>413</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>415</td>
<td>B</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>234</td>
<td>211</td>
<td>A</td>
<td>jones</td>
<td>Forbes</td>
</tr>
</tbody>
</table>

ssn, c-id -> grade
ssn -> name, address

<table>
<thead>
<tr>
<th>Ssn</th>
<th>c-id</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>413</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>415</td>
<td>B</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>234</td>
<td>211</td>
<td>A</td>
<td>jones</td>
<td>Forbes</td>
</tr>
</tbody>
</table>

ssn, c-id -> grade
ssn -> name, address
Overview - detailed

• DB design and normalization
  – pitfalls of bad design
  – decomposition
    • lossless join decomp.
    • dependency preserving
  – normal forms

Decomposition - depend. pres.
informally: we don’t want the original FDs to span two tables - counter-example:

<table>
<thead>
<tr>
<th>S#</th>
<th>address</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>London</td>
<td>E</td>
</tr>
<tr>
<td>125</td>
<td>Paris</td>
<td>E</td>
</tr>
<tr>
<td>234</td>
<td>Pitts.</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S#</th>
<th>address</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>London</td>
<td>E</td>
</tr>
<tr>
<td>125</td>
<td>Paris</td>
<td>E</td>
</tr>
<tr>
<td>234</td>
<td>Pitts.</td>
<td>A</td>
</tr>
</tbody>
</table>

S# -> address, status
address -> status

(Q: Why is it an issue?)

Decomposition - depend. pres.
informally: we don’t want the original FDs to span two tables - counter-example:

<table>
<thead>
<tr>
<th>S#</th>
<th>address</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>London</td>
<td>E</td>
</tr>
<tr>
<td>125</td>
<td>Paris</td>
<td>E</td>
</tr>
<tr>
<td>234</td>
<td>Pitts.</td>
<td>A</td>
</tr>
<tr>
<td>999</td>
<td>Pitts.</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S#</th>
<th>address</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>London</td>
<td>E</td>
</tr>
<tr>
<td>125</td>
<td>Paris</td>
<td>E</td>
</tr>
<tr>
<td>234</td>
<td>Pitts.</td>
<td>A</td>
</tr>
<tr>
<td>999</td>
<td>Pitts.</td>
<td>E</td>
</tr>
</tbody>
</table>

S# -> address, status
address -> status

(Q: Why is it an issue?)

(A: insert [999, Pitts., E])
Decomposition - depend. pres.
informally: we don’t want the original FDs to span two tables - counter-example:

<table>
<thead>
<tr>
<th>S#</th>
<th>address</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>London</td>
<td>E</td>
</tr>
<tr>
<td>125</td>
<td>Paris</td>
<td>E</td>
</tr>
<tr>
<td>234</td>
<td>Pitts.</td>
<td>A</td>
</tr>
</tbody>
</table>

S# -> address, status
address -> status

S# -> status
address -> status

S# -> address, status
address -> status

S# -> address
address -> status

Decomposition - depend. pres.
A subtle point
To avoid it, use the ‘canonical cover’ of the FDs

Decomposition - depend. pres.
dependency preserving decomposition:

<table>
<thead>
<tr>
<th>S#</th>
<th>address</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>London</td>
<td>E</td>
</tr>
<tr>
<td>125</td>
<td>Paris</td>
<td>E</td>
</tr>
<tr>
<td>234</td>
<td>Pitts.</td>
<td>A</td>
</tr>
</tbody>
</table>

S# -> address, status
address -> status

S# -> address
address -> status

S# -> address
address -> status

(but: S#->status ?)
Decomposition - depend. pres.

informally: we don’t want the original FDs to span two tables.
More specifically: … the FDs of the canonical cover.

Q: why is dependency preservation good?

S# -> address
S# -> status
(address->status: 'lost')

Decomposition - depend. pres.
A1: insert [999, Pitts., E] -> REJECT

S# -> address
S# -> status
(address->status: 'lost')

Decomposition - depend. pres.
A2: eg., record that 'Philly' has status ‘A’

S# -> address
S# -> status
(address->status: 'lost')
Decomposition - conclusions

- decompositions should always be lossless
  - joining attribute -> superkey  **MUST HAVE**
- whenever possible, we want them to be dependency preserving (occasionally, impossible - see ‘STJ’ example later…)

Nice to have

Overview - detailed

- DB design and normalization
  - pitfalls of bad design
  - decomposition (-> how to fix the problem)
  - **normal forms** (-> how to detect the problem)
    - BCNF,
    - 3NF
    - (1NF, 2NF)

Normal forms - BCNF

We saw how to fix ‘bad’ schemas - but what is a ‘good’ schema?

Answer: ‘good’, if it obeys a ‘normal form’, ie., a set of rules.

Typically: Boyce-Codd Normal form

Normal forms - BCNF

**Defn.** Rel. $R$ is in BCNF wrt $F$, if

- informally: everything depends on the full key, and nothing but the key

Or

- semi-formally: *every determinant (of the cover) is a candidate key*
Normal forms - BCNF

Example and counter-example:

<table>
<thead>
<tr>
<th>Ssn</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>999</td>
<td>smith</td>
<td>Shady</td>
</tr>
<tr>
<td>234</td>
<td>jones</td>
<td>Forbes</td>
</tr>
</tbody>
</table>

ssn->name, address

<table>
<thead>
<tr>
<th>Ssn</th>
<th>C-id</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>413</td>
<td>A</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>415</td>
<td>B</td>
<td>smith</td>
<td>Main</td>
</tr>
<tr>
<td>234</td>
<td>411</td>
<td>A</td>
<td>jones</td>
<td>Forbes</td>
</tr>
</tbody>
</table>

Drill: Check formal dfn:
- a->b trivial, or
- a is superkey
Normal forms - BCNF

Theorem: given a schema \( R \) and a set of FD ‘\( F \)’, we can always decompose it to schemas \( R_1, \ldots R_n \), so that
- \( R_1, \ldots R_n \) are in BCNF and
- the decompositions are lossless.
(but, some decomp. might lose dependencies)

How? algorithm in book: for a relation \( R \)
- for every FD \( X \rightarrow A \) that violates BCNF,
  decompose to tables (\( X,A \)) and (\( R-A \))
- repeat recursively
eg. TAKES1(\( ssn, c-id, grade, name, address \))
  \( ssn \rightarrow name, address \)
  \( ssn, c-id \rightarrow grade \)

Normal forms - BCNF

eg. TAKES1(\( ssn, c-id, grade, name, address \))
  \( ssn \rightarrow name, address \)
  \( ssn, c-id \rightarrow grade \)
Normal forms - BCNF

pictorially: we want a ‘star’ shape

name

ssn

address

grade

c-id

: not in BCNF

Normal forms - BCNF

or a star-like: (eg., 2 cand. keys):

STUDENT(ssn, st#, name, address)

ssn

name

address

st#

Normal forms - BCNF

but not:

name

ssn

address

st#

Normal forms - BCNF

or

D

e

F

G

H

or

D

E

F

G

H
Overview - detailed

- DB design and normalization
  - pitfalls of bad design
  - decomposition (\(\Rightarrow\) how to fix the problem)
  - normal forms (\(\Rightarrow\) how to detect the problem)
    - BCNF,
    - 3NF
    - (1NF, 2NF)

Normal forms - BCNF

Theorem: given a schema R and a set of FD ‘F’, we can always decompose it to schemas R1, … Rn, so that
- R1, … Rn are in BCNF and
- the decompositions are lossless.
(but, some decomp. might lose dependencies)

Subtle answer

In some rare cases, like the (Student, Teacher, subject) setting:
Normal forms - 3NF

consider the ‘classic’ case:
STJ( Student, Teacher, subJect)
  T -> J
  S,J -> T
is it BCNF?

1) R1(T,J) R2(S,J)
   (BCNF? - lossless? - dep. pres.? )
2) R1(T,J) R2(S,T)
   (BCNF? - lossless? - dep. pres.? )

How to decompose it to BCNF?

1) R1(T,J) R2(S,J)
   (BCNF? Y+Y - lossless? N - dep. pres.? N )
2) R1(T,J) R2(S,T)
   (BCNF? Y+Y - lossless? Y - dep. pres.? N )
Normal forms - 3NF

STJ( Student, Teacher, subJect)  
T-> J  S,J -> T  
in this case: impossible to have both  
• BCNF and  
• dependency preservation  
Welcome 3NF!

Informally, 3NF ‘forgives’ the red arrow in the canonical cover.

Formally, a rel. R with FDs ‘F’ is in 3NF if:  
for every \( a \rightarrow b \) in F:  
• it is trivial or  
• \( a \) is a superkey or  
• \( b \): part of a candidate key

how to bring a schema to 3NF?  
two algo’s in book: First one:  
• start from ER diagram and turn to tables  
• then we have a set of tables \( R_1, \ldots, R_n \) which are in 3NF  
• for each FD \( (X \rightarrow A) \) in the cover that is not preserved, create a table \( (X, A) \)
Normal forms - 3NF

how to bring a schema to 3NF?
two algo’s in book: Second one (‘synthesis’)
• take all attributes of R
• for each FD (X->A) in the cover, add a table (X,A)
• if not lossless, add a table with appropriate key

Example:
R: ABC
F: A->B, C->B
Q1: what is the cover? What is the cand. key?
A1: ‘F’ is the cover; ‘AC’ is the cand. key
Q2: what is the decomposition to 3NF?
A2: R1(A,B), R2(C,B), ...
[is it lossless??]
Normal forms - 3NF

Example:
R: ABC
F: A->B, C->B
Q1: what is the cover? What is the cand. key?
A1: ‘F’ is the cover; ‘AC’ is the cand. key
Q2: what is the decomposition to 3NF?
A2: R1(A,B), R2(C,B), R3(A,C)

Normal forms - 3NF vs BCNF

• If ‘R’ is in BCNF, it is always in 3NF (but not the reverse)
• In practice, aim for
  – BCNF; lossless join; and dep. preservation
• if impossible, we accept
  – 3NF; but insist on lossless join and dep. preservation

Normal forms - more details

• why ‘3’ NF? what is 2NF? 1NF?
• 1NF: attributes are atomic (ie., no set-valued attr., a.k.a. ‘repeating groups’)

<table>
<thead>
<tr>
<th>Ssn</th>
<th>Name</th>
<th>Dependents</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Smith</td>
<td>Peter, Mary, John</td>
</tr>
<tr>
<td>234</td>
<td>Jones</td>
<td>Ann, Michael</td>
</tr>
</tbody>
</table>

not 1NF

Normal forms - more details

2NF: 1NF and non-key attr. fully depend on the key
counter-example: TAKES1(ssn, c-id, grade, name, address)
ssn -> name, address ssn, c-id -> grade

not 2NF
Normal forms - more details

- 3NF: 2NF and no transitive dependencies
- counter-example:

```
A

B

C

D
```
in 2NF, but **not** in 3NF

Normal forms - more details

- 4NF, multivalued dependencies etc: IGNORE
- in practice, E-R diagrams usually lead to tables in BCNF

Overview - conclusions

DB design and normalization
- pitfalls of bad design
- decompositions (lossless, dep. preserving)
- normal forms (BCNF or 3NF)

“everything should depend on the key, the **whole** key, and **nothing** but the key”