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

takes1 (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
  - space
  - (inconsistencies)
  - insertion/deletion anomalies:
Pitfalls

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

<table>
<thead>
<tr>
<th>EID</th>
<th>Class</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Smith</td>
<td>A</td>
<td>Jones</td>
<td>NULL</td>
</tr>
<tr>
<td>34</td>
<td>NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pitfalls

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

<table>
<thead>
<tr>
<th>EID</th>
<th>Class</th>
<th>Grade</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Smith</td>
<td>A</td>
<td>Smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>Smith</td>
<td>B</td>
<td>Smith</td>
<td>Main</td>
</tr>
<tr>
<td>123</td>
<td>Smith</td>
<td>A</td>
<td>Smith</td>
<td>Main</td>
</tr>
</tbody>
</table>

Solution: decomposition

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

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

Decompositions

There are ‘bad’ decompositions. Good ones are:
• lossless and
• dependency preserving

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>
<th>c-id</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>80</td>
<td>John</td>
<td>Smith, Main</td>
<td>171</td>
<td>B</td>
</tr>
<tr>
<td>123</td>
<td>90</td>
<td>Bob</td>
<td>Smith, Main</td>
<td>172</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>85</td>
<td>Jane</td>
<td>Smith, Main</td>
<td>173</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>125</td>
<td>B</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>A</td>
<td>110</td>
<td>A</td>
<td>Main</td>
</tr>
<tr>
<td>125</td>
<td>A</td>
<td>115</td>
<td>B</td>
<td>Main</td>
</tr>
<tr>
<td>234</td>
<td>A</td>
<td>211</td>
<td>A</td>
<td>Forbes</td>
</tr>
</tbody>
</table>

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>D</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>D</td>
</tr>
<tr>
<td>234</td>
<td>Pitts</td>
<td>A</td>
</tr>
</tbody>
</table>

(keep: is it lossless?)

<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>D</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>D</td>
</tr>
<tr>
<td>234</td>
<td>Pitts</td>
<td>A</td>
</tr>
</tbody>
</table>

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

Definition:
consider schema R, with FD 'F'. R1, R2 is a lossless join decomposition of R if we always have: \( r_1 \cap r_2 = r \)

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:
\[
R_1 \cap R_2 \rightarrow R_1 \text{ or } R_1 \cap R_2 \rightarrow R_2
\]

Decomposition - lossless

example:

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
<td>name, address</td>
</tr>
<tr>
<td>c-id</td>
<td>grade</td>
</tr>
<tr>
<td>ssn, c-id -&gt; grade</td>
<td>ssn-&gt;name, address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
<td>name, address</td>
</tr>
<tr>
<td>c-id</td>
<td>grade</td>
</tr>
</tbody>
</table>

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

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

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

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>A</td>
</tr>
<tr>
<td>223</td>
<td>Falls</td>
<td>A</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>A</td>
</tr>
<tr>
<td>223</td>
<td>Falls</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>Pitts</td>
<td>A</td>
</tr>
</tbody>
</table>

S# -> address, status
address -> status

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>A</td>
</tr>
<tr>
<td>223</td>
<td>Falls</td>
<td>A</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>A</td>
</tr>
<tr>
<td>223</td>
<td>Falls</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>Pitts</td>
<td>A</td>
</tr>
</tbody>
</table>

S# -> address, status
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.

why is dependency preservation good?

<table>
<thead>
<tr>
<th>S#</th>
<th>address</th>
<th>address</th>
<th>status</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>London</td>
<td>London</td>
<td>023 A</td>
<td>023 A</td>
</tr>
<tr>
<td>125</td>
<td>Paris</td>
<td>Paris</td>
<td>025 B</td>
<td>025 B</td>
</tr>
<tr>
<td>234</td>
<td>Pitta</td>
<td>Pitta</td>
<td>034 A</td>
<td>034 A</td>
</tr>
</tbody>
</table>

S# -> address  
S# -> status  
(address->status: ‘lost’)

A: eg., record that ‘Philly’ has status ‘A’

<table>
<thead>
<tr>
<th>S#</th>
<th>address</th>
<th>address</th>
<th>status</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>London</td>
<td>London</td>
<td>023 A</td>
<td>023 A</td>
</tr>
<tr>
<td>125</td>
<td>Paris</td>
<td>Paris</td>
<td>025 B</td>
<td>025 B</td>
</tr>
<tr>
<td>234</td>
<td>Pitta</td>
<td>Pitta</td>
<td>034 A</td>
<td>034 A</td>
</tr>
</tbody>
</table>

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

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

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
- semi-formally: every determinant (of the cover) is a candidate key

Example and counter-example:

<table>
<thead>
<tr>
<th>ssn</th>
<th>name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<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-id</th>
<th>grade</th>
<th>name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>84</td>
<td>Smith</td>
<td>Shady</td>
</tr>
<tr>
<td>234</td>
<td>111</td>
<td>Jones</td>
<td>Forbes</td>
</tr>
</tbody>
</table>

ssn -> name, address

ssn, c-id -> grade

ssn, id -> grade

Formally: for every FD \( a \rightarrow b \) in F
- \( a \rightarrow b \) is trivial (\( a \) superset of \( b \)) or
- \( a \) is a 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 decomps. might lose dependencies)

  - for every FD $X \to A$ that violates BCNF,
    decompose to tables $(X, A)$ and $(R-A)$
  - repeat recursively

eg. $\text{TAKES1}(\text{ssn, c-id, grade, name, address})$
  $\text{ssn} \to \text{name, address}$
  $\text{ssn, c-id} \to \text{grade}$

eg. $\text{TAKES1}(\text{ssn, c-id, grade, name, address})$
  $\text{ssn} \to \text{name, address}$
  $\text{ssn, c-id} \to \text{grade}$
Normal forms - BCNF

pictorially: we want a ‘star’ shape

A

B

C

D

E

F

G

H


ssn->name, address

name

ssn->grant

c-id

address

grade

: not in BCNF

ssn

c-id

name

address
Normal forms - BCNF

or a star-like: (eg., 2 cand. keys):
STUDENT(ssn, st#, name, address)

name

address

ssn

st#

name

address

ssn

st#

Normal forms - BCNF

but not:

A

B

D

C

or

D

E

F

G

H

Normal forms - 3NF

consider the ‘classic’ case:
STJ( Student, Teacher, subject)
T -> J
S,J -> T
is it BCNF?
Normal forms - 3NF

STJ( Student, Teacher, subJect)

T-> J   S, J -> T

How to decompose it to BCNF?

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

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

Formally, a rel. R with FDs ‘F’ is in 3NF if:
for every a->b in F:
• it is trivial or
• a is a superkey or
• b: part of a candidate key
Normal forms - 3NF

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 R1, ... Rn which are in 3NF
• for each FD (X->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?
Q2: what is the decomposition to 3NF?
Normal forms - 3NF

Example:
R: ABC
F: A→B, C→B
Q1: what is the cover?
A1: ‘F’ is the cover
Q2: what is the decomposition to 3NF?
A2: R1(A,B), R2(C,B), ...
   [is it lossless??]
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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>John</td>
</tr>
<tr>
<td>234</td>
<td>Jones</td>
<td>Ann</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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
Normal forms - more details

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

```
A
   
D  
   
B  
   
C
```

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”