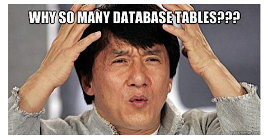
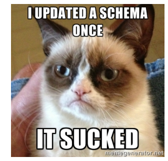
Introduction to Database Systems
CSE 414

Lecture 21: BCNF

What makes good schemas?





Keys

- A **superkey** is a set of attributes $A_1, ..., A_n$ s.t. for any other attribute B, we have $A_1, ..., A_n \rightarrow B$
- A **key** is a minimal superkey (in terms of # of attributes)
  - A superkey and for which no subset is a superkey

Computing (Super)Keys

- For all sets X, compute X^+
- If X^+ = [all attributes], then X is a superkey
- Try reducing to the minimal X's to get the key

Relational Schema Design

SSN → Name, City

Name	SSN	PhoneNumber	City
Fred	123-45-6789	206-555-1234	Seattle
Fred	123-45-6789	206-555-6543	Seattle
Joe	987-65-4321	908-555-2121	Westfield

Anomalies:
- Redundancy = repeat data
- Update anomalies = what if Fred moves to "Bellevue"?
- Deletion anomalies = what if Joe deletes his phone number?

Relation Decomposition

Break the relation into two: SSN → Name, City

Name	SSN	PhoneNumber	City
Fred	123-45-6789	206-555-1234	Seattle
Fred	123-45-6789	206-555-6543	Seattle
Joe	987-65-4321	908-555-2121	Westfield

Name	SSN	City
Fred	123-45-6789	Seattle
Joe	987-65-4321	Westfield

SSN	PhoneNumber
123-45-6789	206-555-1234
123-45-6789	206-555-6543
987-65-4321	908-555-2121

Anomalies have gone:
- No more repeated data
- Easy to move Fred to "Bellevue" (how ?)
- Easy to delete all Joe's phone numbers (how ?)

Eliminating Anomalies

Main idea:

- X → A is OK if X is a (super)key
- X → A is not OK otherwise
  - Need to decompose the table, but how?

Boyce-Codd Normal Form

Boyce-Codd Normal Form

There are no "bad" FDs:

Definition. A relation R is in BCNF if:

Whenever X→ B is a non-trivial dependency, then X is a superkey.

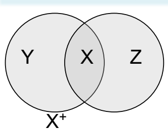
Equivalently:

Definition. A relation R is in BCNF if:

∀ X, either $X^+ = X$ (i.e., X is not in any FDs)
or X^+ = [all attributes] (computed using FDs)

BCNF Decomposition Algorithm

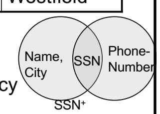
Normalize(R)
find X s.t.: X ≠ X^+ and X^+ ≠ [all attributes]
if (not found) **then** "R is in BCNF"
let Y = X^+ - X; Z = [all attributes] - X^+
decompose R into R1(X ∪ Y) and R2(X ∪ Z)
Normalize(R1); Normalize(R2);



Example

Name	SSN	PhoneNumber	City
Fred	123-45-6789	206-555-1234	Seattle
Fred	123-45-6789	206-555-6543	Seattle
Joe	987-65-4321	908-555-2121	Westfield
Joe	987-65-4321	908-555-1234	Westfield

SSN → Name, City



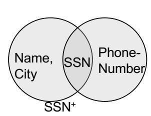
Hence SSN → Name, City is a "bad" dependency

In other words:
SSN+ = SSN, Name, City and is neither SSN nor All Attributes

Example BCNF Decomposition

Name	SSN	City
Fred	123-45-6789	Seattle
Joe	987-65-4321	Westfield

SSN → Name, City



SSN	PhoneNumber
123-45-6789	206-555-1234
123-45-6789	206-555-6543
987-65-4321	908-555-2121
987-65-4321	908-555-1234

Let's check anomalies:
- Redundancy ?
- Update ?
- Delete ?

Find X s.t.: X ≠X^+ and X^+ ≠ [all attributes]

Example BCNF Decomposition

Person(name, SSN, age, hairColor, phoneNumber)
SSN → name, age
age → hairColor

Find X s.t.: X ≠X^+ and X^+ ≠ [all attributes]

Example BCNF Decomposition

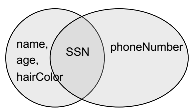
Person(name, SSN, age, hairColor, phoneNumber)

SSN → name, age

age → hairColor

Iteration 1: Person: SSN+ = SSN, name, age, hairColor

Decompose into: P(SSN, name, age, hairColor)
Phone(SSN, phoneNumber)



Find X s.t.: X ≠X^+ and X^+ ≠ [all attributes]

Example BCNF Decomposition

Person(name, SSN, age, hairColor, phoneNumber)

SSN → name, age

age → hairColor

What are the keys ?

Iteration 1: Person: SSN+ = SSN, name, age, hairColor

Decompose into: P(SSN, name, age, hairColor)
Phone(SSN, phoneNumber)

Iteration 2: P: age+ = age, hairColor

Decompose: People(SSN, name, age)
Hair(age, hairColor)
Phone(SSN, phoneNumber)

Find X s.t.: X ≠X^+ and X^+ ≠ [all attributes]

Example BCNF Decomposition

Person(name, SSN, age, hairColor, phoneNumber)

SSN → name, age

age → hairColor

Note the keys!

Iteration 1: Person: SSN+ = SSN, name, age, hairColor

Decompose into: P(<u>SSN</u>, name, age, hairColor)
Phone(<u>SSN, phoneNumber</u>)

Iteration 2: P: age+ = age, hairColor

Decompose: People(<u>SSN</u>, name, age)
Hair(<u>age</u>, hairColor)
Phone(<u>SSN, phoneNumber</u>)

R(A,B,C,D)

Example: BCNF

A → B
B → C

R(A,B,C,D)

R(A,B,C,D)

Example: BCNF

A → B
B → C

Recall: Find X s.t.: X ≠X^+ and X^+ ≠ [all attributes]

R(A,B,C,D)

R(A,B,C,D)

Example: BCNF

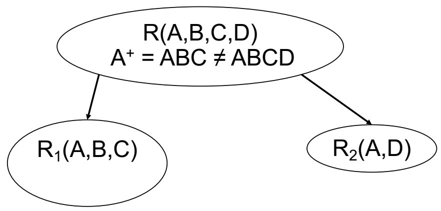
A → B
B → C

R(A,B,C,D)
A^+ = ABC ≠ ABCD

R(A,B,C,D)

Example: BCNF

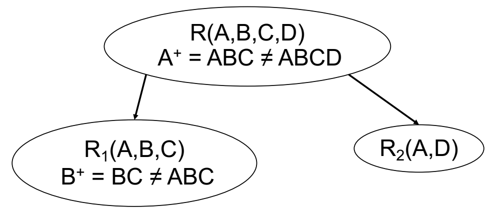
A → B
B → C



R(A,B,C,D)

Example: BCNF

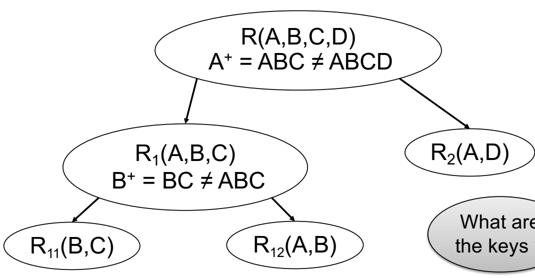
A → B
B → C



R(A,B,C,D)

Example: BCNF

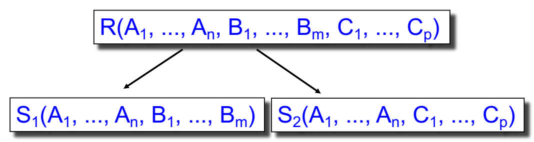
A → B
B → C



What are the keys ?

What happens if in R we first pick B^+ ? Or AB^+ ?

Decompositions in General



S_1 = projection of R on $A_1, ..., A_n, B_1, ..., B_m$
S_2 = projection of R on $A_1, ..., A_n, C_1, ..., C_p$

Lossless Decomposition

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Name	Price
Gizmo	19.99
OneClick	24.99
~~Gizmo~~	~~19.99~~

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Lossy Decomposition

What is lossy here?

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Price	Category
19.99	Gadget
24.99	Camera
19.99	Camera

Lossy Decomposition

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Price	Category
19.99	Gadget
24.99	Camera
19.99	Camera

Decomposition in General

$R(A_1, \ldots, A_n, B_1, \ldots, B_m, C_1, \ldots, C_p)$

$S_1(A_1, \ldots, A_n, B_1, \ldots, B_m)$ $S_2(A_1, \ldots, A_n, C_1, \ldots, C_p)$

Let: S_1 = projection of R on $A_1, \ldots, A_n, B_1, \ldots, B_m$
S_2 = projection of R on $A_1, \ldots, A_n, C_1, \ldots, C_p$

The decomposition is called *lossless* if $R = S_1 \bowtie S_2$

Fact: If $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ then the decomposition is lossless

It follows that every BCNF decomposition is lossless

Schema Refinements = Normal Forms

- 1st Normal Form = all tables are flat
- 2nd Normal Form = obsolete
- Boyce Codd Normal Form = no bad FDs
- 3rd Normal Form = see book
  - BCNF is lossless but can cause loss of ability to check some FDs (see book 3.4.4)
  - 3NF fixes that (is lossless and dependency-preserving), but some tables might not be in BCNF – i.e., they may have redundancy anomalies

Getting Practical

How to implement normalization in SQL

Motivation

- We learned about how to normalize tables to avoid anomalies
- How can we implement normalization in SQL if we can't modify existing tables?
  - This might be due to legacy applications that rely on previous schemas to run

Use Views!

- A view in SQL =
  - A table computed from other tables, s.t., whenever the base tables are updated, the view is updated too
- More generally:
  - A view is derived data that keeps track of changes in the original data

Purchase(customer, product, store)
Product(pname, price)
StorePrice(store, price)

A Simple View

Create a view that returns for each store the prices of products purchased at that store

```
CREATE VIEW StorePrice AS
  SELECT DISTINCT x.store, y.price
  FROM Purchase AS x, Product AS y
  WHERE x.product = y.pname
```

This is like a new table StorePrice(store,price)

Purchase(customer, product, store)
Product(pname, price)
StorePrice(store, price)

We Use a View Like Any Table

- A "high end" store is a store that sell some products over 1000.
- For each customer, return all the high end stores that they visit.

```
SELECT DISTINCT u.customer, u.store
FROM Purchase AS u, StorePrice AS v
WHERE u.store = v.store
   AND v.price > 1000
```


Types of Views

- Virtual views
  - Computed only on-demand – slow at runtime
  - Always up to date
- Materialized views
  - Pre-computed offline – fast at runtime
  - May have stale data (must recompute or update)
- A key component of database performance tuning is the selection of materialized and virtual views

Vertical Partitioning

Resumes

SSN	Name	Address	Resume	Picture
234234	Mary	Houston	Doc1…	JPG1…
345345	Sue	Seattle	Doc2…	JPG2…
345343	Joan	Seattle	Doc3…	JPG3…
432432	Ann	Portland	Doc4…	JPG4…

T1

SSN	Name	Address
234234	Mary	Houston
345345	Sue	Seattle
. . .		

T2

SSN	Resume
234234	Doc1…
345345	Doc2…

T3

SSN	Picture
234234	JPG1…
345345	JPG2…

T2.SSN is a key *and* a foreign key to **T1**.SSN. Same for **T3**.SSN

T1(ssn,name,address)
T2(ssn,resume)
T3(ssn,picture)
Resumes(ssn,name,address,resume,picture)

Vertical Partitioning

```
CREATE VIEW Resumes AS
  SELECT T1.ssn, T1.name, T1.address,
         T2.resume, T3.picture
  FROM   T1,T2,T3
  WHERE  T1.ssn=T2.ssn AND T1.ssn=T3.ssn
```


T1(ssn,name,address)
T2(ssn,resume)
T3(ssn,picture)
Resumes(ssn,name,address,resume,picture)

Vertical Partitioning

```
CREATE VIEW Resumes AS
  SELECT T1.ssn, T1.name, T1.address,
         T2.resume, T3.picture
  FROM   T1,T2,T3
  WHERE  T1.ssn=T2.ssn AND T1.ssn=T3.ssn
```

```
SELECT address
FROM   Resumes
WHERE  name = 'Sue'
```

T1(ssn,name,address)
T2(ssn,resume)
T3(ssn,picture)

Resumes(ssn,name,address,resume,picture)

Vertical Partitioning

```
CREATE VIEW  Resumes  AS
   SELECT  T1.ssn, T1.name, T1.address,
           T2.resume, T3.picture
   FROM    T1,T2,T3
   WHERE  T1.ssn=T2.ssn AND T1.ssn=T3.ssn
```

```
SELECT address
FROM    Resumes
WHERE   name = 'Sue'
```

Original query:

```
SELECT T1.address
FROM T1, T2, T3
WHERE T1.name = 'Sue'
      AND T1.SSN=T2.SSN
      AND T1.SSN = T3.SSN
```


T1(ssn,name,address)
T2(ssn,resume)
T3(ssn,picture)

Resumes(ssn,name,address,resume,picture)

Vertical Partitioning

```
CREATE VIEW  Resumes  AS
   SELECT  T1.ssn, T1.name, T1.address,
           T2.resume, T3.picture
   FROM    T1,T2,T3
   WHERE  T1.ssn=T2.ssn AND T1.ssn=T3.ssn
```

```
SELECT address
FROM    Resumes
WHERE   name = 'Sue'
```

Modified query:

SELECT T1.address
FROM T1, ~~T2, T3~~
WHERE T1.name = 'Sue'
~~AND T1.SSN=T2.SSN~~
~~AND T1.SSN = T3.SSN~~

Final query:

```
SELECT T1.address
FROM T1
WHERE T1.name = 'Sue'
```

Vertical Partitioning Applications

- Advantages
  - Speeds up queries that touch only a small fraction of columns
  - Single column can be compressed effectively, reducing disk I/O
- Disadvantages
  - Updates are expensive!
  - Need many joins to access many columns
  - Repeated key columns add overhead