Design Theory

BBM471 Database Management Systems
Dr. Fuat Akal
akal@hacettepe.edu.tr

Design Theory I

Today's Lecture

1. Normal forms & functional dependencies
2. Finding functional dependencies
3. Closures, superkeys & keys

1. Normal forms & functional dependencies

What you will learn about in this section

1. Overview of design theory & normal forms
2. Data anomalies & constraints
3. Functional dependencies

Design Theory

- Design theory is about how to represent your data to avoid ***anomalies***.
- It is a mostly mechanical process
  - Tools can carry out routine portions

Normal Forms

- 1st Normal Form (1NF) = All tables are flat
- *2nd Normal Form = disused*
- **Boyce-Codd Normal Form (BCNF)**
- **3rd Normal Form (3NF)**

DB designs based on *functional dependencies*, intended to prevent data ***anomalies***

Our focus in this lecture

- *4th and 5th Normal Forms = see text books*

1st Normal Form (1NF)

Student	Courses
Mary	{CS145,CS229}
Joe	{CS145,CS106}
…	…

Violates 1NF.

Student	Courses
Mary	CS145
Mary	CS229
Joe	CS145
Joe	CS106

In 1st NF

1NF Constraint: Types must be atomic!

Data Anomalies & Constraints

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

If every course is in only one room, contains ***redundant*** information!

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes *anomalies*:

Student	Course	Room
Mary	CS145	B01
Joe	CS145	C12
Sam	CS145	B01
..	..	..

If we update the room number for one tuple, we get inconsistent data = an *update* anomaly

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes *anomalies*:

Student	Course	Room
..	..	..

If everyone drops the class, we lose what room the class is in! = a *delete* anomaly

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

	CS229	C12

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Similarly, we can't reserve a room without students = an ***insert*** anomaly

Constraints Prevent (some) Anomalies in the Data

Student	Course
Mary	CS145
Joe	CS145
Sam	CS145
..	..

Course	Room
CS145	B01
CS229	C12

Is this form better?

- Redundancy?
- Update anomaly?
- Delete anomaly?
- Insert anomaly?

Today: develop theory to understand why this design may be better **and** how to find this *decomposition*…

Functional Dependencies

Functional Dependency

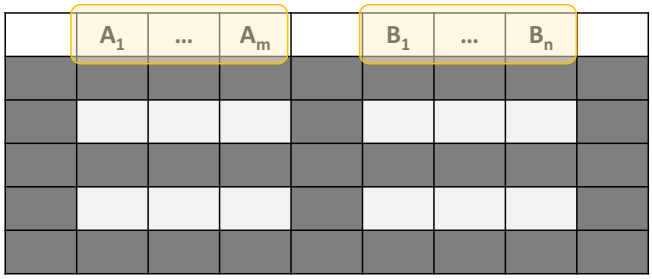
Def: Let A,B be *sets* of attributes
We write A → B or say A ***functionally determines*** B if, for any tuples t_1 and t_2:

$$t_1[A] = t_2[A] \text{ implies } t_1[B] = t_2[B]$$

and we call A → B a <u>functional dependency</u>

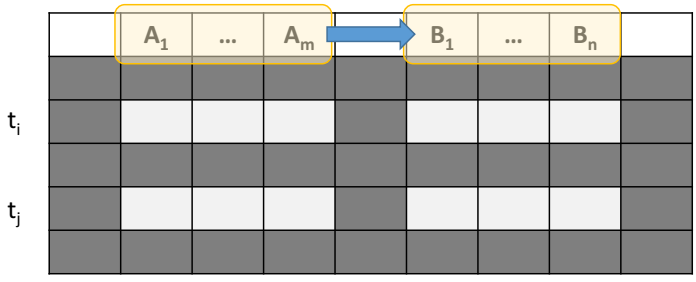
In another words: A->B means that
"whenever two tuples agree on A then they agree on B."

A Picture Of FDs



Defn (again):
Given attribute sets $A=\{A_1,\ldots,A_m\}$ and $B = \{B_1,\ldots B_n\}$ in R,

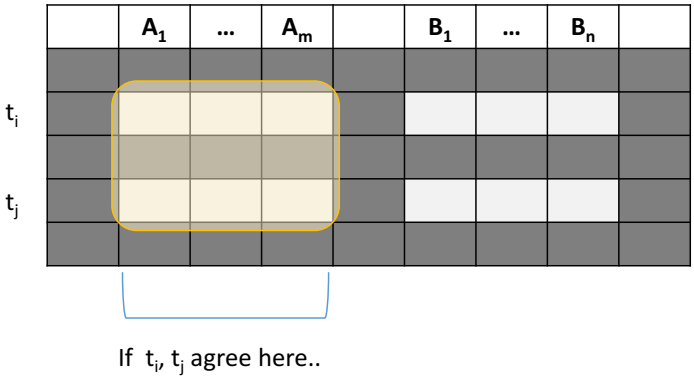
A Picture Of FDs



Defn (again):
Given attribute sets $A=\{A_1,\ldots,A_m\}$ and $B = \{B_1,\ldots B_n\}$ in R,

The *functional dependency* A→ B on R holds if for *any* t_i,t_j in R:

A Picture Of FDs

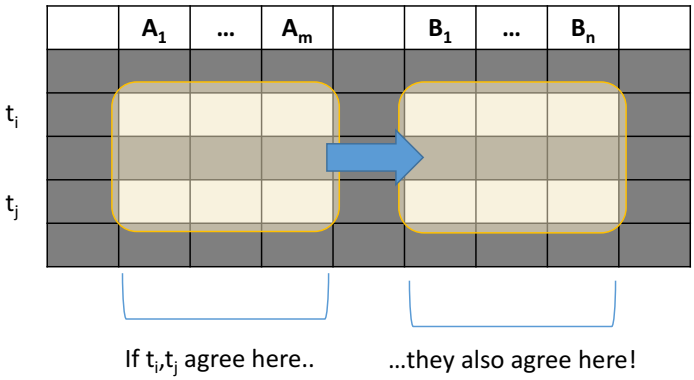


Defn (again):
Given attribute sets $A=\{A_1,\ldots,A_m\}$ and $B = \{B_1,\ldots B_n\}$ in R,

The *functional dependency* A→ B on R holds if for *any* t_i,t_j in R:

$t_i[A_1] = t_j[A_1]$ AND $t_i[A_2]=t_j[A_2]$ AND … AND $t_i[A_m] = t_j[A_m]$

A Picture Of FDs



Defn (again):
Given attribute sets $A=\{A_1,\ldots,A_m\}$ and $B = \{B_1,\ldots B_n\}$ in R,

The *functional dependency* A→ B on R holds if for *any* t_i,t_j in R:

<u>if</u> $t_i[A_1] = t_j[A_1]$ AND $t_i[A_2]=t_j[A_2]$ AND … AND $t_i[A_m] = t_j[A_m]$

<u>then</u> $t_i[B_1] = t_j[B_1]$ AND $t_i[B_2]=t_j[B_2]$ AND … AND $t_i[B_n] = t_j[B_n]$

FDs for Relational Schema Design

- High-level idea: **why do we care about FDs?**

1. Start with some relational *schema*
2. Model its *functional dependencies (FDs)*
3. Use these to *design a better schema*
   - One which minimizes the possibility of anomalies

Functional Dependencies as Constraints

A **functional dependency** is a form of **constraint**

- *Holds* on some instances not others.
- Part of the schema, helps define a valid *instance*.

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Note: The FD {Course} -> {Room} ***holds on this instance***

*Recall: an **instance** of a schema is a multiset of tuples conforming to that schema, **i.e. a table***

Functional Dependencies as Constraints

Note that:

- You can check if an FD is **violated** by examining a single instance;
- However, you **cannot prove** that an FD is part of the schema by examining a single instance.
  - *This would require checking every valid instance*

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

However, cannot *prove* that the FD {Course} -> {Room} is ***part of the schema***

More Examples

An FD is a constraint which holds, or does not hold on an instance:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

More Examples

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876 ←	Salesrep
E1111	Smith	9876 ←	Salesrep
E9999	Mary	1234	Lawyer

{Position} → {Phone}

More Examples

EmpID	Name	Phone	Position
E0045	Smith	1234 →	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234 →	Lawyer

but *not* {Phone} → {Position}

ACTIVITY

A	B	C	D	E
1	2	4	3	6
3	2	5	1	8
1	4	4	5	7
1	2	4	3	6
3	2	5	1	8

Find at least *three* FDs which are violated on this instance:

{ } → { }
{ } → { }
{ } → { }

2. Finding functional dependencies

What you will learn about in this section

1. "Good" vs. "Bad" FDs: Intuition
2. Finding FDs
3. Closures

"Good" vs. "Bad" FDs

We can start to develop a notion of **good** vs. **bad** FDs:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

Intuitively:

EmpID -> Name, Phone, Position is *"good FD"*

- ***Minimal redundancy, less possibility of anomalies***

“Good” vs. “Bad” FDs

We can start to develop a notion of **good** vs. **bad** FDs:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

Intuitively:

EmpID -> Name, Phone, Position is *“good FD”*

Position -> Phone *is a “bad FD”*

- ***Redundancy! Possibility of data anomalies***

“Good” vs. “Bad” FDs

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Returning to our original example… can you see how the “bad FD” {Course} -> {Room} could lead to an:

- Update Anomaly
- Insert Anomaly
- Delete Anomaly
- …

Given a set of FDs (from user) our goal is to:

1. Find all FDs, and
2. Eliminate the “Bad Ones".

FDs for Relational Schema Design

- High-level idea: **why do we care about FDs?**

1. Start with some relational *schema*
2. Find out its *functional dependencies (FDs)*
3. Use these to *design a better schema*
   - One which minimizes possibility of anomalies

This part can be tricky!

Finding Functional Dependencies

- There can be a very **large number** of FDs…
  - *How to find them all efficiently?*
- We can't necessarily show that any FD will hold **on all instances…**
  - *How to do this?*

We will start with this problem:
Given a set of FDs, F, what other FDs ***must*** hold?

Finding Functional Dependencies

Equivalent to asking: Given a set of FDs, F = $\{f_1,\ldots f_n\}$, does an FD g hold?

Inference problem: How do we decide?

Finding Functional Dependencies

Example:

Products

Name	Color	Category	Dep	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Department}
3. {Color, Category} → {Price}

Given the provided FDs, we can see that **{Name, Category} → {Price}** must also hold on **any instance**…

Which / how many other FDs do?!?

Finding Functional Dependencies

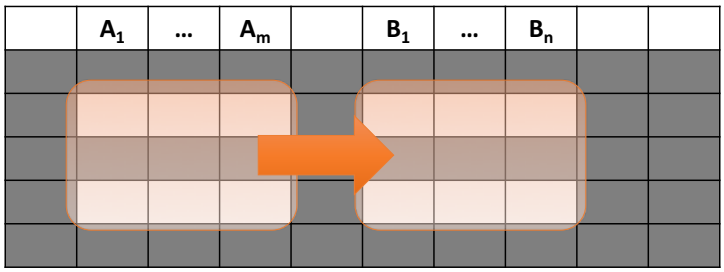
Equivalent to asking: Given a set of FDs, F = {f_1,…f_n}, does an FD g hold?

Inference problem: How do we decide?

Answer: Three simple rules called **Armstrong's Rules.**

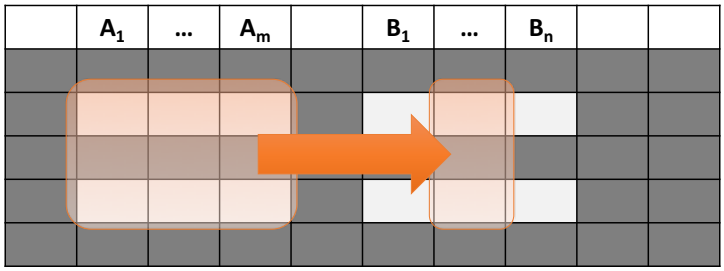
1. Split/Combine,
2. Reduction (Trivial), and
3. Transitivity… *ideas by picture*

1. Split/Combine



A_1, …, A_m → B_1,…,B_n

1. Split/Combine

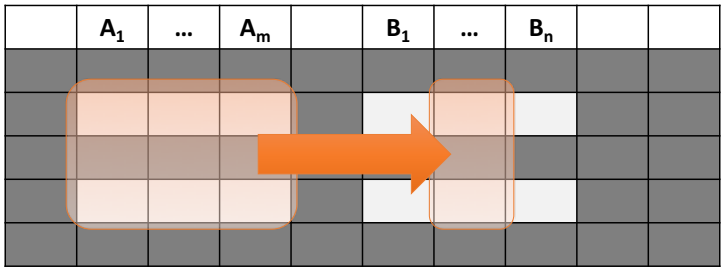


$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

… is equivalent to the following *n* FDs…

$A_1,\ldots,A_m \rightarrow B_i$ for i=1,…,n

1. Split/Combine

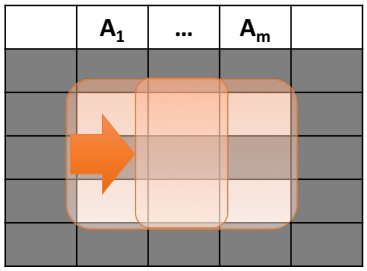


And vice-versa, $A_1,\ldots,A_m \rightarrow B_i$ for i=1,…,n

… is equivalent to …

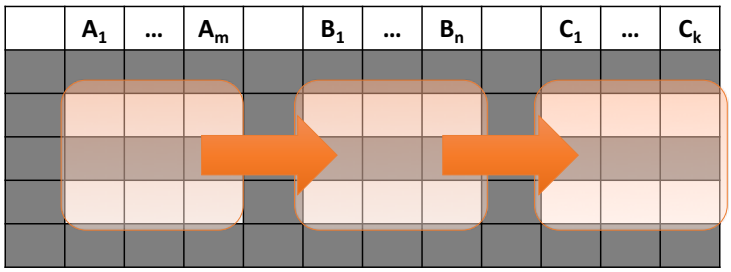
$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

Reduction/Trivial



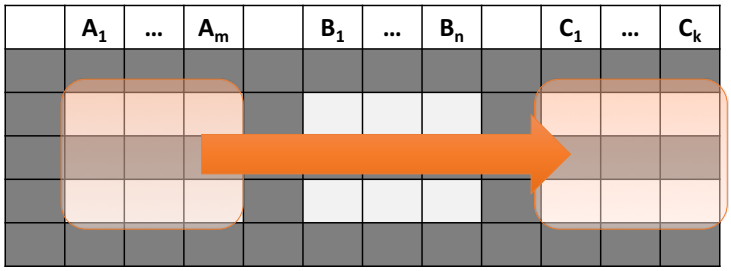
$A_1,\ldots,A_m \rightarrow A_j$ for any j=1,…,m

3. Transitive Closure



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$ and
$B_1,\ldots,B_n \rightarrow C_1,\ldots,C_k$

3. Transitive Closure



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$ and
$B_1,\ldots,B_n \rightarrow C_1,\ldots,C_k$

implies

$A_1,\ldots,A_m \rightarrow C_1,\ldots,C_k$

Finding Functional Dependencies

Example:

Products

Name	Color	Category	Dept	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept}
3. {Color, Category} → {Price}

Which / how many other FDs hold?

Finding Functional Dependencies

Example:

Inferred FDs:

Inferred FD	Rule used
4. {Name, Category} -> {Name}	?
5. {Name, Category} -> {Color}	?
6. {Name, Category} -> {Category}	?
7. {Name, Category -> {Color, Category}	?
8. {Name, Category} -> {Price}	?

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept.}
3. {Color, Category} → {Price}

Which / how many other FDs hold?

Finding Functional Dependencies

Example:

Inferred FDs:

Inferred FD	Rule used
4. {Name, Category} -> {Name}	Trivial
5. {Name, Category} -> {Color}	Transitive (4 -> 1)
6. {Name, Category} -> {Category}	Trivial
7. {Name, Category -> {Color, Category}	Split/combine (5 + 6)
8. {Name, Category} -> {Price}	Transitive (7 -> 3)

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept.}
3. {Color, Category} → {Price}

Can we find an algorithmic way to do this?

Closures

Closure of a set of Attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs **F**:
Then the **closure**, $\{A_1, \ldots, A_n\}^+$ is the set of attributes **B** s.t. $\{A_1, \ldots, A_n\} \rightarrow B$

Example: F =

```
{name} → {color}
{category} → {dept}
{color, category} → {price}
```

Example Closures:

```
{name}+ = {name, color}
{name, category}+ =
{name, category, color, dept, price}
{color}+ = {color}
```

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$ and set of FDs F.

Repeat until X doesn't change; **do**:

if $\{B_1, \ldots, B_n\} \rightarrow$ C is entailed by F

and $\{B_1, \ldots, B_n\} \subseteq$ X

then add C to X.

Return X as X^+

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
if $\{B_1, \ldots, B_n\} \rightarrow$ C is in F **and** $\{B_1, \ldots, B_n\} \subseteq$ X:
then add C to X.
Return X as X^+

`{name, category}+ = {name, category}`

F =

`{name} → {color}`

`{category} → {dept}`

`{color, category} → {price}`

Closure Algorithm

Start with X = {A_1, …, A_n}, FDs F.
Repeat until X doesn't change; **do**:
  if {B_1, …, B_n} → C is in F **and** {B_1, …, B_n} ⊆ X:
    then add C to X.
Return X as X^+

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

{name, category}$^+$ =
{name, category}

{name, category}$^+$ =
{name, category, color}

Closure Algorithm

Start with X = {A_1, …, A_n}, FDs F.
Repeat until X doesn't change; **do**:
  if {B_1, …, B_n} → C is in F **and** {B_1, …, B_n} ⊆ X:
    then add C to X.
Return X as X^+

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

{name, category}$^+$ =
{name, category}

{name, category}$^+$ =
{name, category, color}

{name, category}$^+$ =
{name, category, color, dept}

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

{name, category}$^+$ =
{name, category}

{name, category}$^+$ =
{name, category, color}

{name, category}$^+$ =
{name, category, color, dept}

{name, category}$^+$ =
{name, category, color, dept, price}

Example

R(A,B,C,D,E,F)

{A,B} → {C}
{A,D} → {E}
{B} → {D}
{A,F} → {B}

Compute $\{A,B\}^+ = \{A, B,$ $\}$

Compute $\{A, F\}^+ = \{A, F,$ $\}$

Example

R(A,B,C,D,E,F)

{A,B} → {C}
{A,D} → {E}
{B} → {D}
{A,F} → {B}

Compute $\{A,B\}^+$ = {A, B, C, D }

Compute $\{A, F\}^+$ = {A, F, B }

Example

R(A,B,C,D,E,F)

{A,B} → {C}
{A,D} → {E}
{B} → {D}
{A,F} → {B}

Compute $\{A, B\}^+$ = {A, B, C, D, E}

Compute $\{A, F\}^+$ = {A, B, C, D, E, F}

3. Closures, Superkeys & Keys

What you will learn about in this section

1. Closures Pt. II
2. Superkeys & Keys

Why Do We Need the Closure?

- With closure we can find all FD's easily
- To check if $X \rightarrow A$
  1. Compute X^+
  2. Check if $A \in X^+$

Note here that **X** is a *set* of attributes, but **A** is a *single* attribute. Why does considering FDs of this form suffice?

Recall the Split/combine rule:
$X \rightarrow A_1, \ldots, X \rightarrow A_n$
implies
$X \rightarrow \{A_1, \ldots, A_n\}$

Using Closure to Infer ALL FDs

Step 1: Compute X^+, for every set of attributes X:

Example:
Given F =

{A,B} → C
{A,D} → B
{B} → D

$\{A\}^+ = \{A\}$
$\{B\}^+ = \{B,D\}$
$\{C\}^+ = \{C\}$
$\{D\}^+ = \{D\}$
$\{A,B\}^+ = \{A,B,C,D\}$
$\{A,C\}^+ = \{A,C\}$
$\{A,D\}^+ = \{A,B,C,D\}$
$\{A,B,C\}^+ = \{A,B,D\}^+ = \{A,C,D\}^+ = \{A,B,C,D\}$
$\{B,C,D\}^+ = \{B,C,D\}$
$\{A,B,C,D\}^+ = \{A,B,C,D\}$

Using Closure to Infer ALL FDs

Example:
Given F =

{A,B} → C
{A,D} → B
{B} → D

Step 1: Compute X^+, for every set of attributes X:

{A}⁺ = {A}, {B}⁺ = {B,D}, {C}⁺ = {C}, {D}⁺ = {D}, {A,B}⁺ = {A,B,C,D}, {A,C}⁺ = {A,C}, {A,D}⁺ = {A,B,C,D}, {A,B,C}⁺ = {A,B,D}⁺ = {A,C,D}⁺ = {A,B,C,D}, {B,C,D}⁺ = {B,C,D}, {A,B,C,D}⁺ = {A,B,C,D}

Step 2: Enumerate all FDs X → Y, s.t. $Y \subseteq X^+$ and $X \cap Y = \emptyset$:

{A,B} → {C,D}, {A,D} → {B,C},
{A,B,C} → {D}, {A,B,D} → {C},
{A,C,D} → {B}

Using Closure to Infer ALL FDs

Example:
Given F =

{A,B} → C
{A,D} → B
{B} → D

Step 1: Compute X^+, for every set of attributes X:

{A}⁺ = {A}, {B}⁺ = {B,D}, {C}⁺ = {C}, {D}⁺ = {D}, {A,B}⁺ = {A,B,C,D}, {A,C}⁺ = {A,C}, {A,D}⁺ = {A,B,C,D}, {A,B,C}⁺ = {A,B,D}⁺ = {A,C,D}⁺ = {A,B,C,D}, {B,C,D}⁺ = {B,C,D}, {A,B,C,D}⁺ = {A,B,C,D}

Step 2: Enumerate all FDs X → Y, s.t. $Y \subseteq X^+$ and $X \cap Y = \emptyset$:

"Y is in the closure of X"

{A,B} → {C,D}, {A,D} → {B,C},
{A,B,C} → {D}, {A,B,D} → {C},
{A,C,D} → {B}

Using Closure to Infer ALL FDs

Example:
Given F =

{A,B} → C
{A,D} → B
{B} → D

Step 1: Compute X^+, for every set of attributes X:

$\{A\}^+ = \{A\}$, $\{B\}^+ = \{B,D\}$, $\{C\}^+ = \{C\}$, $\{D\}^+ = \{D\}$, $\{A,B\}^+ = \{A,B,C,D\}$, $\{A,C\}^+ = \{A,C\}$, $\{A,D\}^+ = \{A,B,C,D\}$, $\{A,B,C\}^+ = \{A,B,D\}^+ = \{A,C,D\}^+ = \{A,B,C,D\}$, $\{B,C,D\}^+ = \{B,C,D\}$, $\{A,B,C,D\}^+ = \{A,B,C,D\}$

Step 2: Enumerate all FDs X → Y, s.t. $Y \subseteq X^+$ and $X \cap Y = \varnothing$:

The FD X → Y is non-trivial

{A,B} → {C,D}, {A,D} → {B,C},
{A,B,C} → {D}, {A,B,D} → {C},
{A,C,D} → {B}

Superkeys and Keys

Keys and Superkeys

A **superkey** is a set of attributes $A_1, \ldots, A_n$ s.t. for *any other* attribute **B** in R, we have $\{A_1, \ldots, A_n\} \rightarrow B$

i.e. all attributes are *functionally determined* by a superkey.

Note: superkey is short for super set of keys.

A **key** is a *minimal* superkey

Meaning that no subset of a key is also a superkey

Finding Keys and Superkeys

- For each set of attributes X
  1. Compute X^+
  2. If X^+ = set of all attributes then X is a **superkey**
  3. If X is minimal, then it is a **key**

Do we need to check all sets of attributes? Which sets?

Example of Finding Keys

```
Product(name, price, category, color)
```

```
{name, category} → price
{category} → color
```

What is a key?

Example of Keys

```
Product(name, price, category, color)
```

```
{name, category} → price
{category} → color
```

$\{name, category\}^+ = \{name, price, category, color\}$
= the set of all attributes
$\Longrightarrow$ this is a **superkey**
$\Longrightarrow$ this is a **key**, since neither `name` nor `category` alone is a superkey

Design Theory II

Today's Lecture

1. Boyce-Codd Normal Form
2. Decompositions & 3NF
3. MVDs

1. Boyce-Codd Normal Form

What you will learn about in this section

1. Conceptual Design
2. Boyce-Codd Normal Form
3. The BCNF Decomposition Algorithm

Conceptual Design

Back to Conceptual Design

Now that we know how to find FDs, it's a straight-forward process:

1. Search for "bad" FDs
2. If there are any, then *keep decomposing the table into sub-tables* until no more bad FDs
3. When done, the database schema is *normalized*

Prime vs Non-Prime Attributes

- **Prime Attributes**
  - Attribute set that belongs to any candidate key are called Prime Attributes.
- **Non-Prime Attribute**
  - Attribute set does not belong to any candidate key are called Non-Prime Attributes.

Normal Forms

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce Codd Normal Form (BCNF)

First Normal Form (1NF)

For a table to be in the First Normal Form, it should follow the following 4 rules:

1. It should only have single (atomic) valued attributes/columns.
2. Values stored in a column should be of the same domain
3. All the columns in a table should have unique names.
4. And the order in which data is stored, does not matter.

First Normal Form (1NF) ?

id	name	course
1	Fuat	BBM471 BBM467 CMP652
2	Zeynep	BBM101 BBM102

UNF – Unnormalized Form
or, NF2

Second Normal Form (2NF)

For a table to be in the Second Normal Form

1. It should be in the First Normal form.
2. No non-prime attribute is dependent on any proper subset of any candidate key of the relation, i.e., *no partial dependency*.
   - All non-prime attributes are fully functional dependent on the primary key

Second Normal Form (2NF) ?

id	name	course	course name
1	Fuat	BBM471	Database Management Systems
1	Fuat	BBM467	Data Intensive Apps
1	Fuat	CMP652	Next Gen DBMS
2	Zeynep	BBM101	Programming I
2	Zeynep	BBM102	Programming II

Second Normal Form (2NF) ?

TABLE_PURCHASE_DETAIL

Customer ID	Store ID	Purchase Location
1	1	Los Angeles
1	3	San Francisco
2	1	Los Angeles
3	2	New York
4	3	San Francisco

This table has a composite primary key [Customer ID, Store ID]. The non-key attribute is [Purchase Location].

In this case, [Purchase Location] only depends on [Store ID], which is only part of the primary key.

TABLE_PURCHASE

Customer ID	Store ID
1	1
1	3
2	1
3	2
4	3

TABLE_STORE

Store ID	Purchase Location
1	Los Angeles
2	New York
3	San Francisco

https://www.1keydata.com/database-normalization/second-normal-form-2nf.php

Third Normal Form (3NF)

A table is said to be in the Third Normal Form when

1. It is in the Second Normal form.
2. All the attributes in a table are determined only by the candidate keys of that relation and not by any non-prime attributes, i.e., *no Transitive Dependency*.
   - There is no transitive functional dependency

Third Normal Form (3NF) ?

id	name	city	country
1	Fuat	Ankara	Turkey
2	Zeynep	Columbus	USA
3	Dila	Zurich	Switzerland

Third Normal Form (3NF) ?

TABLE_BOOK_DETAIL

Book ID	Genre ID	Genre Type	Price
1	1	Gardening	25.99
2	2	Sports	14.99
3	1	Gardening	10.00
4	3	Travel	12.99
5	2	Sports	17.99

[Book ID] determines [Genre ID], and [Genre ID] determines [Genre Type].

Therefore, [Book ID] determines [Genre Type] via [Genre ID] and we have transitive functional dependency, and this structure does not satisfy third normal form.

TABLE_BOOK

Book ID	Genre ID	Price
1	1	25.99
2	2	14.99
3	1	10.00
4	3	12.99
5	2	17.99

TABLE_GENRE

Genre ID	Genre Type
1	Gardening
2	Sports
3	Travel

https://www.1keydata.com/database-normalization/third-normal-form-3nf.php

Boyce-Codd Normal Form (BCNF)

- Main idea is that we define "good" and "bad" FDs as follows:
  - X → A is a *"good FD" if X is a (super)key*
    - In other words, if A is the set of all attributes
  - X → A is a *"bad FD"* otherwise
- We will try to eliminate the "bad" FDs!

Boyce-Codd Normal Form (BCNF)

- Why does this definition of "good" and "bad" FDs make sense?
- If X is *not* a (super)key, it functionally determines *some* of the attributes
  - *Recall*: this means there is redundancy
  - And redundancy like this can lead to data anomalies!

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

Boyce-Codd Normal Form

BCNF is a simple condition for removing anomalies from relations:

A relation R is **in BCNF** if:

if $\{A_1, ..., A_n\} \rightarrow B$ is a *non-trivial* FD in R

then $\{A_1, ..., A_n\}$ **is a superkey** for R

Equivalently: ∀ sets of attributes X, either ($X^+ = X$) or (X^+ = all attributes)

In other words: there are no "bad" FDs

Boyce-Codd Normal Form

If all non-key attributes depend upon only the complete key, the relation is in BCNF.

Informally the Boyce-Codd normal form is expressed as "*Each attribute must represent a fact about the key, the whole key, and nothing but the key.*"

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}

This FD is *bad* because it is **not** a superkey

⟹ **Not** in BCNF

What is the key?
{SSN, PhoneNumber}

Example

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

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

{SSN} → {Name,City}

This FD is now *good* because it is the key

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

Now in BCNF!

Example

Author	Nationality	Book title	Genre	Number of pages
William Shakespeare	English	The Comedy of Errors	Comedy	100
Markus Winand	Austrian	SQL Performance Explained	Textbook	200
Jeffrey Ullman	American	A First Course in Database Systems	Textbook	500
Jennifer Widom	American	A First Course in Database Systems	Textbook	500

Author	Nationality
William Shakespeare	English
Markus Winand	Austrian
Jeffrey Ullman	American
Jennifer Widom	American

Book title	Genre	Number of pages
The Comedy of Errors	Comedy	100
SQL Performance Explained	Textbook	200
A First Course in Database Systems	Textbook	500

Author	Book title
William Shakespeare	The Comedy of Errors
Markus Winand	SQL Performance Explained
Jeffrey Ullman	A First Course in Database Systems
Jennifer Widom	A First Course in Database Systems

BCNF Decomposition Algorithm

BCNFDecomp(R):

BCNF Decomposition Algorithm

BCNFDecomp(R):
  Find *a set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

Find a set of attributes X which has non-trivial "bad" FDs, i.e. is not a superkey, using closures

BCNF Decomposition Algorithm

BCNFDecomp(R):
  Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

  if (not found) **then Return** R

If no "bad" FDs found, in BCNF!

BCNF Decomposition Algorithm

BCNFDecomp(R):
  Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

  if (not found) then Return R

  let $Y = X^+ - X$, $Z = (X^+)^C$

Let Y be the attributes that ***X functionally determines*** (+ that are not in X)

And let Z be **the other attributes that it *doesn't***

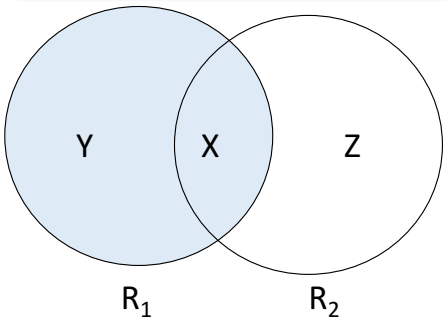
BCNF Decomposition Algorithm

BCNFDecomp(R):
  Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

  if (not found) then Return R

  let $Y = X^+ - X$, $Z = (X^+)^C$
  decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

Split into one relation (table) with X plus the attributes that X determines (Y)…



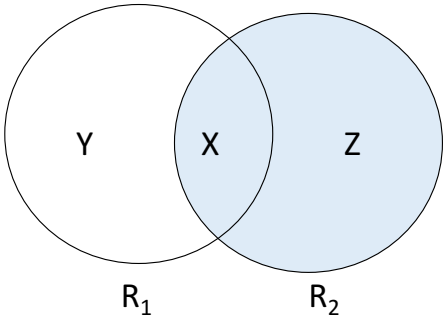
BCNF Decomposition Algorithm

BCNFDecomp(R):
  Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

  if (not found) then Return R

  let $Y = X^+ - X$, $Z = (X^+)^C$
  decompose R into $\mathbf{R_1(X \cup Y)}$ and $\mathbf{R_2(X \cup Z)}$

And one relation with X plus the attributes it *does not* determine (Z)



BCNF Decomposition Algorithm

BCNFDecomp(R):
  Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

  if (not found) then Return R

  let $Y = X^+ - X$, $Z = (X^+)^C$
  decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

  Return BCNFDecomp(R_1), BCNFDecomp(R_2)

Proceed recursively until no more “bad” FDs!

Example

BCNFDecomp(R):
Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) then Return R

let $Y = X^+ - X$, $Z = (X^+)^C$
decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

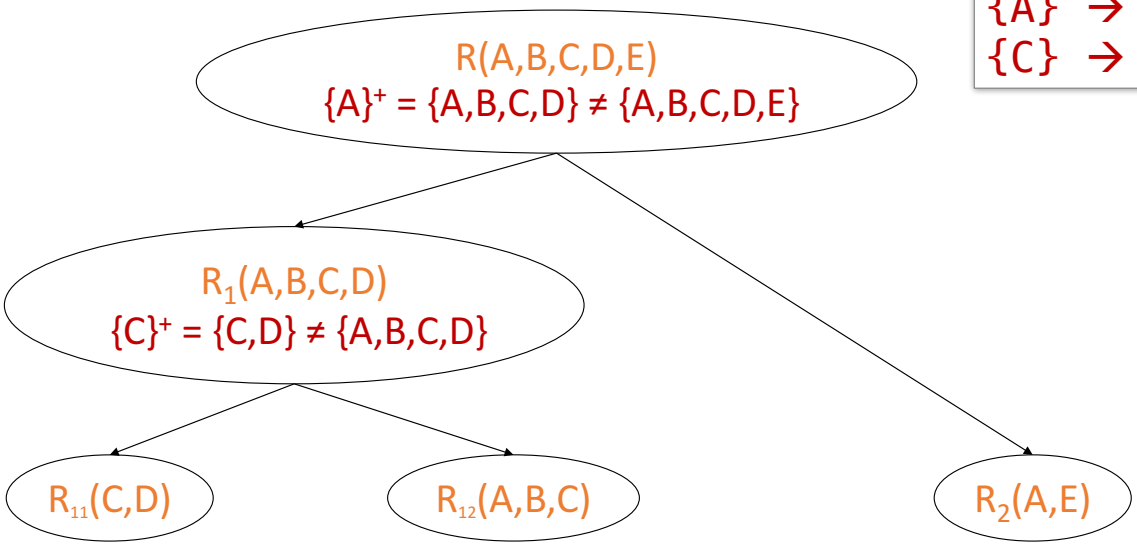
R(A,B,C,D,E)

{A} → {B,C}
{C} → {D}

Example

R(A,B,C,D,E)

{A} → {B,C}
{C} → {D}



2. Decompositions

Recap: Decompose to remove redundancies

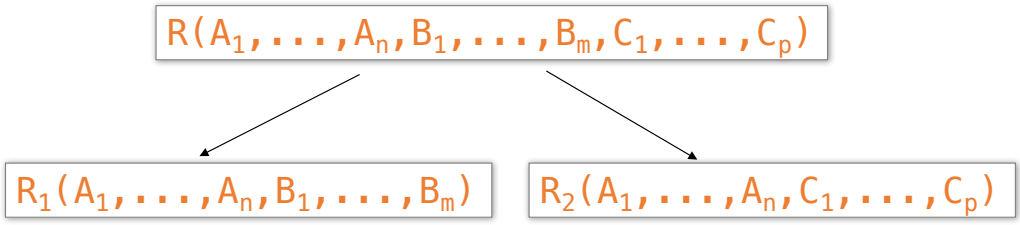
1. We saw that **redundancies** in the data ("bad FDs") can lead to data anomalies

2. We developed mechanisms to **detect and remove redundancies by decomposing tables into BCNF**
   1. BCNF decomposition is *standard practice*- very powerful & widely used!

3. However, sometimes decompositions can lead to **more subtle unwanted effects…**

When does this happen?

Decompositions in General



R_1 = the *projection* of R on A_1, ..., A_n, B_1, ..., B_m

R_2 = the *projection* of R on A_1, ..., A_n, C_1, ..., C_p

Theory of 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

Sometimes a decomposition is "correct"

i.e. it is a Lossless decomposition

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

However sometimes it isn't

What's wrong here?

Lossless Decompositions

$R(A_1,...,A_n,B_1,...,B_m,C_1,...,C_p)$

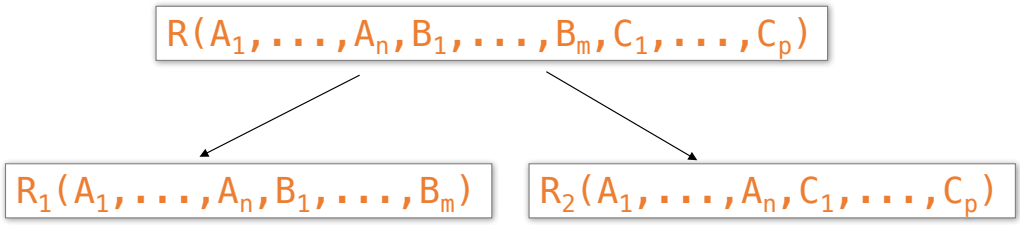
$R_1(A_1,...,A_n,B_1,...,B_m)$ $R_2(A_1,...,A_n,C_1,...,C_p)$

What (set) relationship holds between R1 Join R2 and R if lossless?

Hint: Which tuples of R will be present?

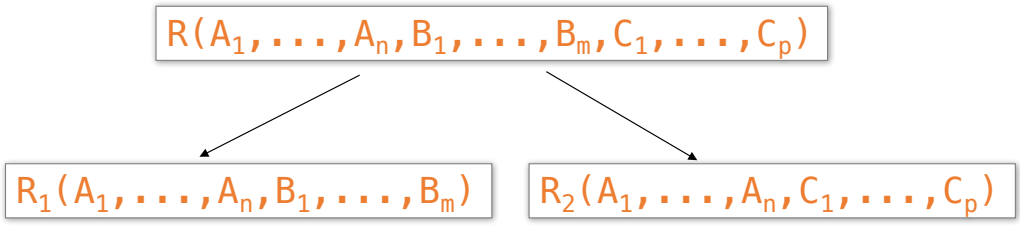
It's lossless if we have equality!

Lossless Decompositions



A decomposition R to (R1, R2) is **lossless** if R = R1 Join R2

Lossless Decompositions



If $\{A_1, ..., A_n\} \rightarrow \{B_1, ..., B_m\}$
Then the decomposition is lossless

Note: don't need
$\{A_1, ..., A_n\} \rightarrow \{C_1, ..., C_p\}$

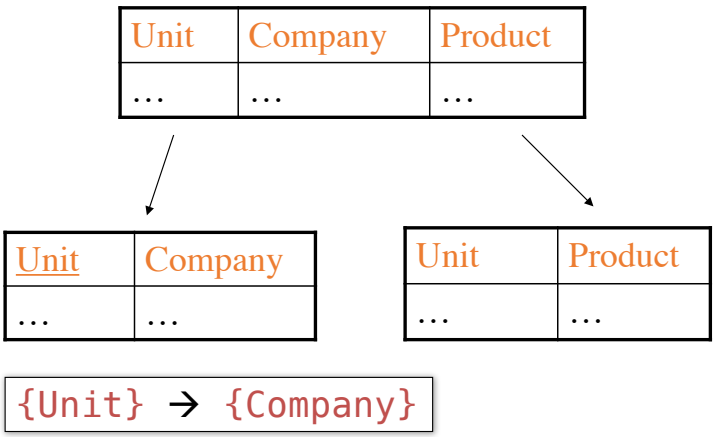
BCNF decomposition is always lossless. Why?

A problem with BCNF

Problem: To enforce a FD, must reconstruct original relation—*on each insert!*

Note: This is historically inaccurate, but it makes it easier to explain

A Problem with BCNF



`{Unit}` → `{Company}`

`{Unit}` → `{Company}`
`{Company,Product}` → `{Unit}`

We do a BCNF decomposition on a "bad" FD:
$\{Unit\}^+ = \{Unit, Company\}$

We lose the FD `{Company,Product}` → `{Unit}`!!

So Why is that a Problem?

Unit	Company
Galaga99	UW
Bingo	UW

Unit	Product
Galaga99	Databases
Bingo	Databases

No problem so far. All *local* FD's are satisfied.

{Unit} → {Company}

Unit	Company	Product
Galaga99	UW	Databases
Bingo	UW	Databases

Let's put all the data back into a single table again:

Violates the FD {Company,Product} → {Unit}!!

The Problem

- We started with a table R and FDs F
- We decomposed R into BCNF tables R_1, R_2, … with their own FDs F_1, F_2, …
- We insert some tuples into each of the relations—which satisfy their local FDs but when reconstruct it violates some FD **across** tables!

Practical Problem: To enforce FD, must reconstruct R—*on each insert!*

Possible Solutions

- Various ways to handle so that decompositions are all lossless / no FDs lost
  - For example 3NF- stop short of full BCNF decompositions.
- Usually a tradeoff between redundancy / data anomalies and FD preservation…

BCNF still most common- with additional steps to keep track of lost FDs…

3. MVDs

What you will learn about in this section

1. MVDs

Multi-Value Dependencies (MVDs)

- A multi-value dependency (MVD) is another type of dependency that could hold in our data, ***which is not captured by FDs***

- Formal definition:
  - Given a relation **R** having attribute set **A**, and two sets of attributes $\mathbf{X}, \boldsymbol{Y} \subseteq \boldsymbol{A}$
  - The ***multi-value dependency (MVD)*** $\boldsymbol{X} \twoheadrightarrow \boldsymbol{Y}$ holds on R if
  - ***for any tuples*** $\boldsymbol{t_1}, \boldsymbol{t_2} \in \boldsymbol{R}$ s.t. $\boldsymbol{t_1}[\boldsymbol{X}] = \boldsymbol{t_2}[\boldsymbol{X}]$**,** there exists a tuple $\boldsymbol{t_3}$ s.t.:
    - $\mathbf{t_1[X] = t_2[X] = t_3[X]}$
    - $\mathbf{t_1[Y] = t_3[Y]}$
    - $\mathbf{t_2[A \setminus Y] = t_3[A \setminus Y]}$
      - *Where A \ B means "elements of set A not in set B"*

Multi-Value Dependencies (MVDs)

- One less formal, literal way to phrase the definition of an MVD:
- ***The MVD $X \twoheadrightarrow Y$*** holds on R if for any pair of tuples with the same X values, the "swapped" pair of tuples with the same X values, but the other permutations of Y and A\Y values, is also in R

Ex: X = {x}, Y = {y}:

x	y	z
1	0	1
1	1	0

For $X \twoheadrightarrow Y$ to hold must have…

x	y	z
1	0	1
1	1	0
1	0	0
1	1	1

Note the connection to a local *cross-product…*

Multi-Value Dependencies (MVDs)

- Another way to understand MVDs, in terms of *conditional independence:*
- ***The MVD $X \twoheadrightarrow Y$*** holds on R if given X, Y is conditionally independent of A \ Y and vice versa…

Here, given x = 1, we know for ex. that:
y = 0 → z = 1

I.e. z is conditionally ***dependent*** on y given x

x	y	z
1	0	1
1	1	0

Here, this is not the case!

I.e. z is conditionally ***independent*** of y given x

x	y	z
1	0	1
1	1	0
1	0	0
1	1	1

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

Are there any functional dependencies that might hold here?

No…

And yet it seems like there is some pattern / dependency…

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre…

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre…

Given a set of movies and snacks…

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre…

Given a set of movies and snacks…

Any movie / snack combination is possible!

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write **{A} ↠ {B}** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write **{A} ↠ {B}** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$
- and $t_3[R\backslash B] = t_2[R\backslash B]$

Where R\B is "R minus B" i.e. the attributes of R not in B

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_2	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
t_3	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_1	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

Note this also works!

Remember, an MVD holds over *a relation or an instance*, so defn. must hold for every applicable pair…

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_2	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
t_3	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_1	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

This expresses a sort of dependency (= data redundancy) that we *can't* express with FDs

**Actually, it expresses conditional independence (between film and snack given movie theatre)!*

Example

Students

Student_Name	Major	Sport
Ravi	Art History	Soccer
Ravi	Art History	Volleyball
Ravi	Art History	Tennis
Beth	Chemistry	Tennis
Beth	Chemistry	Soccer

This is read as "Student_Name multidetermines Major" and "Student_Name multidetermines Sport."

Students & Majors

Student_Name	Major
Ravi	Art History
Ravi	Art History
Ravi	Art History
Beth	Chemistry
Beth	Chemistry

Students & Sports

Student_Name	Sport
Ravi	Soccer
Ravi	Volleyball
Ravi	Tennis
Beth	Tennis
Beth	Soccer

Summary

- Normal forms describe how to **remove** this redundancy by **decomposing** relations
- Normalization let us avoid anomalies
- Normalization saves space, but space is cheap
- Normalization simplifies updates, but read are more common
- Normalization does not help in optimizing queries much

Acknowledgements

The course material used for this lecture is mostly taken and/or adopted from the course materials of the *CS145 Introduction to Databases* lecture given by *Christopher Ré* at *Stanford University* (http://web.stanford.edu/class/cs145/).