CS 6400 A

# Database Systems Concepts and Design

Lecture 6
09/14/26

# Announcements

Assignment 1 due this Wednesday (Sep 16)

In-class midterm next Wednesday (Sep 23)

- Contexts covered: lec 2 (SQL I) – lec 7 (Design Theory II)
- Review lecture next Monday (Sep 21)
- Format: open book and notes (as long as they are printed)

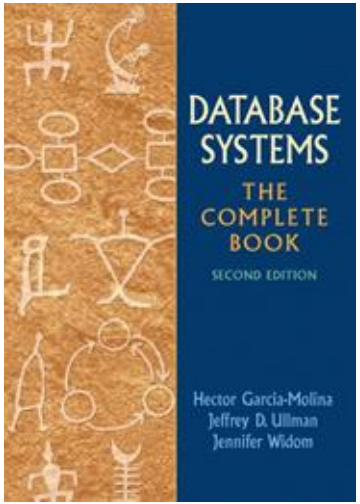
Remember to sign up on Canvas for project!

- Use the Piazza post to find teammates!

# Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 3: Design Theory for Relational Databases (3.1 – 3.3)




Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang and CS145 (Intro to Big Data Systems) taught by Peter Bailis

# Agenda

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

# 1. Normal forms & functional dependencies

# Normal Forms

- 1st Normal Form (1NF) = All tables are flat
- 2nd Normal Form = disused
- Boyce-Codd Normal Form (BCNF)
- 3rd Normal Form (3NF)
- 4th and 5th Normal Forms = see textbooks

# 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!

# 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

- 4th and 5th Normal Forms = see textbooks

Our focus in this lecture + next one

# Data Anomalies

A poorly designed database causes *anomalies*:

| Student | Course | Room |
|---|---|---|
| Mary | CS6400 | B01 |
| Joe | CS6400 | B01 |
| Sam | CS6400 | B01 |
| .. | .. | .. |

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

# Data Anomalies

A poorly designed database causes *anomalies*:

| Student | Course | Room |
|---|---|---|
| Mary | CS6400 | B01 |
| Joe | CS6400 | C12 |
| Sam | CS6400 | B01 |
| .. | .. | .. |

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

# Data Anomalies

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

# Data Anomalies

A poorly designed database causes *anomalies*:

| | | |
|---|---|---|
| … | CS6422 | C12 |



| Student | Course | Room |
|---|---|---|
| Mary | CS6400 | B01 |
| Joe | CS6400 | B01 |
| Sam | CS6400 | B01 |
| .. | .. | .. |

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

# Data Anomalies

| Student | Course |
|---|---|
| Mary | CS6400 |
| Joe | CS6422 |
| Sam | CS6400 |
| .. | .. |

| Course | Room |
|---|---|
| CS6400 | B01 |
| CS6422 | C12 |

Eliminate anomalies by **decomposing relations**.

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

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

# ER Diagram vs Design Theory

Typical workflow: Requirements → ER diagram → Relational schema → Design theory → Refined schema

Both help us design good databases, from different starting points.

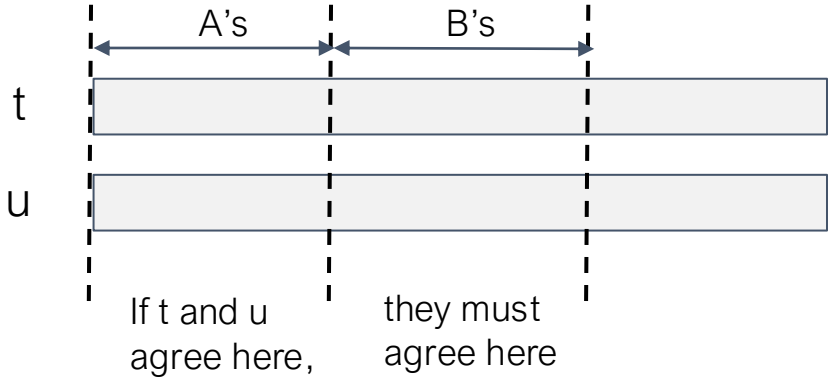
| ER | Design Theory |
|---|---|
| Start from application requirements | Start from a relational schema + constraints |
| What should the database represent? | Is this a good way to organize the data? |
| Identify entities, relationships, attributes | Identify dependencies among attributes |

# Functional Dependencies

# Functional dependency (FD)

**Definition**: if two tuples of R agree on all the attributes $A_1$, $A_2$, …, $A_n$, they must also agree on (or functionally determine) $B_1$, $B_2$, …, $B_m$

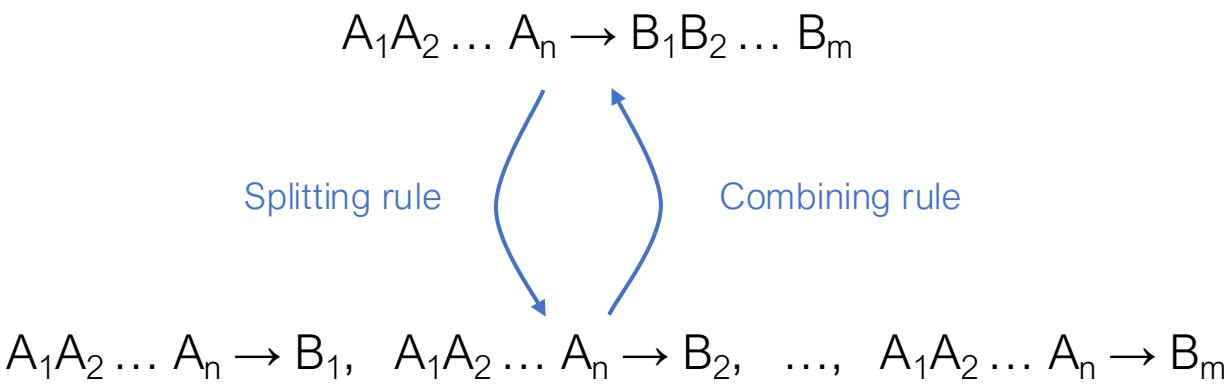
- Denoted as $A_1A_2 \ldots A_n \rightarrow B_1B_2 \ldots B_m$




A->B means that "whenever two tuples agree on A then they agree on B."

# Splitting/combining rule

- Splitting/combining can be applied to the **right sides** of FD's



# Splitting/combining rule

- For example,

title year → length genre studioName



title year → length
title year → genre
title year → studioName

# Splitting rule

- Splitting rule does not apply to the left sides of FD's

title year → length



title → length
year → length

# Functional Dependencies as Constraints

A functional dependency is a form of constraint

- Part of the schema, helps define a valid instance
- Holds on some *instances* (but not others) – can check whether there are violations

| Student | Course | Room |
|---|---|---|
| Mary | CS6400 | B01 |
| Joe | CS6400 | B01 |
| Sam | CS6400 | B01 |
| .. | .. | .. |

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

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

# Functional Dependencies: Instance vs Schema

Given a particular relation instance:

- If the FD is **violated**, we can tell immediately from the data.
- If the FD is satisfied, we cannot conclude that it is part of the schema (it may hold only by coincidence).
  - An FD is a constraint on *all valid instances* of the relation.

| Student | Course | Room |
|---|---|---|
| Mary | CS6400 | B01 |
| Joe | CS6400 | B01 |
| Sam | CS6400 | B01 |
| .. | .. | .. |

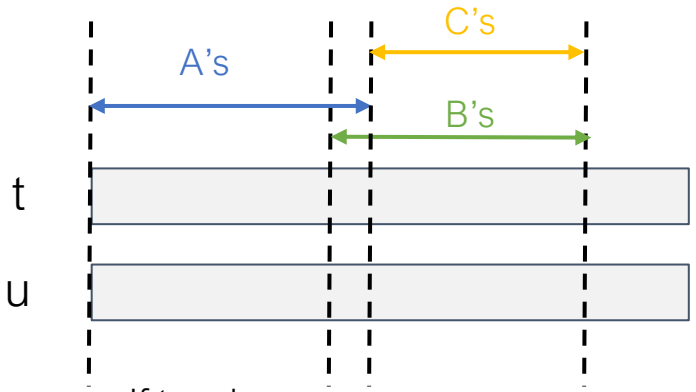
However, cannot prove that the FD {Course} -> {Room} is part of the schema

# Trivial functional dependencies

A constraint is *trivial* if it holds for every possible instance of the relation.

**Trivial FDs:**
$A_1A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$ such that
$\{B_1, B_2, \ldots B_m\} \subseteq \{A_1, A_2, \ldots, A_n\}$

**Trivial dependency rule:**
$A_1A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$ is equivalent to $A_1A_2 \ldots A_n \rightarrow C_1\ C_2 \ldots C_k$, where the C's are the B's that are not also A's




If t and u agree on the As

Then they must agree on the Bs

And surely they agree on the Cs

# In-class Exercise

Q1: Find an FD that holds on this instance
Q2: Find an FD that is violated on this instance

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

# 2. Finding functional dependencies

# 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
    1. One which minimizes possibility of anomalies

This part can be tricky!

# Finding Functional Dependencies

There can be a large number of FDs…

Let's start with this problem:
Given a set of FDs, $F = \{f_1,\ldots f_n\}$, does an FD $g$ hold?

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

1. Reflexivity,
2. Augmentation,
3. Transitivity

# Armstrong's axioms

You can derive any FDs that follows from a given set using these axioms:

1. **Reflexivity**:
   If Y is a subset of X, then $X \rightarrow Y$

This means that a set of attributes always determines a subset of itself

2. **Augmentation**:
   If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z

This means we can add the same attributes to both sides of a functional dependency.

3. **Transitivity**:
   If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

This allows us to chain functional dependencies.

# Armstrong's axioms

- Does AB → D follow from the FDs below?

AB → C

BC → AD

D → E

CF → B

1. AB → C (given)
2. BC → AD (given)

# Armstrong's axioms

- Does AB → D follow from the FDs below?

AB → C

BC → AD

D → E

CF → B

1. AB → C (given)
2. BC → AD (given)
3. AB → BC (Augmentation on 1)

# Armstrong's axioms

- Does AB → D follow from the FDs below?

AB → C

BC → AD

D → E

CF → B

1. AB → C (given)
2. BC → AD (given)
3. AB → BC (Augmentation on 1)
4. AB → AD (Transitivity on 2,3)

# Armstrong's axioms

- Does AB → D follow from the FDs below?

AB → C

BC → AD

D → E

CF → B

1. AB → C (given)
2. BC → AD (given)
3. AB → BC (Augmentation on 1)
4. AB → AD (Transitivity on 2,3)
5. AD → D (Reflexivity)

# Armstrong's axioms

- Does AB → D follow from the FDs below?

AB → C

BC → AD

D → E

CF → B

1. AB → C (given)
2. BC → AD (given)
3. AB → BC (Augmentation on 1)
4. AB → AD (Transitivity on 2,3)
5. AD → D (Reflexivity)
6. AB → D (Transitivity on 4,5)

Can we find an algorithmic way to do this?

# Closures

# Closure of attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs F, the closure, $\{A_1, \ldots, A_n\}^+$ is the set of attributes B where $\{A_1, \ldots, A_n\} \rightarrow B$ follows from the FDs in F

AB → C

BC → AD

D → E

CF → B

{A, B}+

A, B

# Closure of attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs F, the closure, $\{A_1, \ldots, A_n\}^+$ is the set of attributes B where $\{A_1, \ldots, A_n\} \rightarrow$ B follows from the FDs in F

AB → C

BC → AD

D → E

CF → B

{A, B}+

A, B, C

# Closure of attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs F, the closure, $\{A_1, \ldots, A_n\}^+$ is the set of attributes B where $\{A_1, \ldots, A_n\} \rightarrow$ B follows from the FDs in F

AB → C

BC → AD

D → E

CF → B

{A, B}+

A, B, C, D

# Closure of attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs F, the closure, $\{A_1, \ldots, A_n\}^+$ is the set of attributes B where $\{A_1, \ldots, A_n\} \rightarrow$ B follows from the FDs in F

AB → C

BC → AD

D → E

CF → B

{A, B}+

A, B, C, D, E

# Closure of attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs F, the closure, $\{A_1, \ldots, A_n\}^+$ is the set of attributes B where $\{A_1, \ldots, A_n\} \rightarrow$ B follows from the FDs in F

AB → C

BC → AD

D → E

CF → B

{A, B}+

A, B, C, D, E

Cannot be expanded further, so this is a closure

# Closure algorithm

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

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

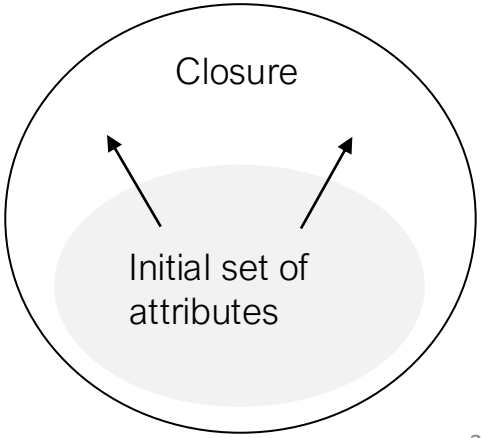
**if** $\{B_1, \ldots, B_n\} \rightarrow C$ is entailed by F

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

**then** add C to X.

**Return** X as $X^+$

Helps to split the FD's of F, so each FD has a single attribute on the right




The algorithm (proof in book)

- only produces true FDs
- discovers all true FDs

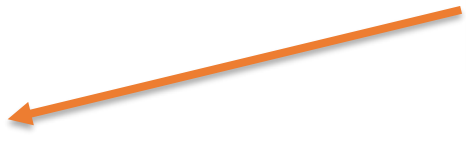
# 3. Closures, Superkeys & Keys

# 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

Example:

Given F =

{A,B} → C
{A,D} → B
{B} → D

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

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

# 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 $X \cap Y = \varnothing$:

# 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:

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

*Y is in the closure of X*

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}

*The FD X → Y is non-trivial*

# Minimal basis

The full set of implied FDs can be large and redundant...

For the purpose of data normalization, it's often easier to work with the cleanest, smallest set of FDs.

Given a set of FD's F, any set of FD's equivalent to F is a **basis** for F

A minimal basis (or minimal cover) for a set of FDs F is a simplified set of FDs G that satisfies the following conditions:

- No redundant/extraneous FDs
- RHS has a single attribute
- No extraneous attributes on the LHS

# Minimal basis generation algorithm

Example Input: F = {A → AB, AB → C}

1. Split FD's so that they have singleton right sides
   G = {A → B, A → A, AB → C}
2. Remove trivial FDs
   G = {A → B, AB → C}
3. Minimize the left sides of each FD
   G = {A → B, A → C}
4. Remove redundant FDs
   G = {A → B, A → C}

Step 3:

*For each FD X → A in F:*
*For each attribute B in X:*
*If (X - {B})+ contains A,*
*remove B from X.*

# Why Do We Need the Closure?

With closure we can find **keys** and **superkeys** of a relation

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**

# 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

A key is a minimal superkey

This means that no subset of a key is also a superkey
(i.e., dropping any attribute from the key makes it no longer a superkey)

# Example of Finding Keys

Product(name, price, category, color)

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

What is a key?

# Example of Finding Keys

Product(name, price, category, color)

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

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

⟹ this is a **superkey**
⟹ this is a **key**, since neither name nor category alone is a superkey

# In-class Exercise

Given R(A, B, C, D) and FD's AB → C, C → D, D → A

- What are all keys of R?

# Back to Design Theory

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*

Recall: there are several normal forms…

# 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 next lecture

4th and 5th Normal Forms = see text books

# 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!