Cs5530/6530
Exam 2 Review

Juliana Freire

Our Objectives in this Course

- First course on database systems: cover the *fundamental database concepts*
- Learn the practical benefits that stem from using a Database Management System (DBMS)
- Use a state-of-the-art DBMS, from designing the schema to loading data and implementing queries
- Cover some advanced topics, including: XML data management and information integration

This course is

- NOT A tutorial on how to use any specific DBMS
  - You will learn the *foundations*, and if you know the foundations you should be able to use *any* DBMS
- NOT A tutorial on SQL
  - You will learn some SQL and *relational algebra,* the algebraic query language that forms the basis for SQL
- NOT A course on database implementation: You will not implement a database system from scratch
  - You will learn how to *use* databases

Practice vs. Theory

- Aim for a balance
  - GOAL assignments: theory (and some practice)
  - Projects: practice and hands-on experience
- I hope this course was effective and that it helped you learn *fundamental database concepts as well as practical benefits of databases*
- Advanced Topics: motivate you to think about different problems and research!

What will be covered?

Everything we covered through last week. Emphasis will be on material not covered in Exam 1:

Functional dependencies

Normalization

XML and Semi-Structured data

Exam Format

- Open book, closed everything else
- Similar to homework assignments

Functional Dependencies

Relational Database Design

- Use the Entity Relationship (ER) model to reason about your data---structure and relationships, then translate model into a relational schema (more on this later)
- Specify relational schema directly
  - Like what you do when you design the data structures for a program

Pitfalls in Relational Database Design

- Relational database design requires that we find a "good" collection of relation schemas
- A bad design may lead to
  - Repetition of Information
  - Inability to represent certain information
- Design Goals:
  - Avoid redundant data
  - Ensure that relationships among attributes are represented
  - Facilitate the checking of updates for violation of database integrity constraints

Functional Dependencies (FDs) and Database Design

- A FD is yet another kind of integrity constraint
- Vital for the redesign of database schemas to *eliminate* redundancy
  - Enable systematic improvement of database designs
- A functional dependency (FD) on relation R is a statement of the form:

$$A1, A2, \ldots, An \rightarrow B$$

If two tuples of R agree on attributes A1, A2, …, An, then they must also agree on some other attribute B

If a relation *R* is legal under a set *F* of FDs, we say *R satisfies F, or that F holds in R*

FD: Example

EMP(ENAME, <u>SSN</u>, STARTDATE,ADDRESS, PHONE)

SSN → ENAME

SSN → STARTDATE

SSN → ADDRESS

SSN → PHONE

Given a SSN, we expect there is a unique employee

- Shorthand:

  SSN → ENAME, STARTDATE, ADDRESS, PHONE

- Do the following FDs hold?

  ENAME → SSN

  PHONE → SSN

What is functional in a functional dependency?

A1,…,An → B

A FD is a *function* that takes a list of values (A1,…,An) and produces a unique value B or no value at all (this value can be the NULL value)

x	f(x)
1	2
2	5
4	5

x	g(x)
1	2
2	2
3	5

x	h(x)
1	10
2	20
3	30

We are looking for functional relationships (that must occur in a relation) among attribute values

What is functional in a functional dependency?

A1,…,An → B

There is a function that takes a li

and produces a unique value B c

this value can be the NUL

Unlike *mathematical* functions, you cannot *compute* from first principles – you need to do it by *looking up in a table*

x	f(x)
1	2
2	5
4	5

x	g(x)
1	2
2	2
3	5

x	h(x)
1	10
2	20
3	30

We are looking for functional relationships
(that must occur in a relation) among attribute values

FDs and Database Instances

PHONE → SSN?

- Which FDs hold for the following table?

ENAME	SSN	START_DATE	PHONE
John	123-45-6789	1999-05-27	
Mary	321-54-9876	1975-03-05	
Melissa	987-65-4321	1985-03-05	

FDs, like any constraint, tell us about the *schema*

- What about for this other table?

ENAME	SSN	START_DATE	PHONE
John	123-45-6789	1999-05-27	9085837689
Mary	321-54-9876	1975-03-05	9085837689
Melissa	987-65-4321	1985-03-05	5057899999

Keys and FDs

EMPLOYEE (SSN, NAME, RATING, HOURLY_WAGE, JOB_DESC)

What does it mean to be a key?

The key attributes uniquely identify the tuple.

For one value of the key, there is only one value for all the other attributes.

There is an FD from the key to every other attribute in the table.

SSN → NAME, RATING, HOURLY_WAGE, JOB_DESC

Minimality of Keys

- While we suppose designers will keep their design *simple,* we have no way of knowing whether a key is minimal
- FDs allow us to reason about the minimality of keys!
- Minimal: you can't throw anything out
- Minimum: the smallest possible
  - E.g., {city,state} and {zipcode} are both (minimal) keys, while {zipcode} is minimum

FDs and Redundancy

- Functional dependencies allow us to express constraints that cannot be expressed using keys

EMPLOYEE (SSN, NAME, RATING, HOURLY_WAGE, JOB_DESC)

rating → hourly_wages

Redundant storage of rating-wage associations

- *Having formal techniques to identify the problem with this design and guide us to a better design is very useful!*

How can FDs help?

- They help remove redundancy by identifying parts into which a relation can be decomposed
  - *E.g.,* $\quad$ *rating* $\rightarrow$ *hourly_wages*

    ssn $\rightarrow$ *name, job_desc*

Bad! EMPLOYEE (<u>SSN</u>, NAME, RATING, HOURLY_WAGE, JOB_DESC)

Good! EMPLOYEE (<u>SSN</u>, NAME, RATING, JOB_DESC)
RATING_WAGE(<u>RATING</u>,HOURLY_WAGE)

Normalization and Reasoning About Functional Dependencies

Functional Dependencies and Normalization

- FDs are the basis of normalization -- a *formal* methodology for *refining and creating good relational designs*
- Normalization places some constraints on the schema to:
  - Reduce redundancy
  - Alleviate update anomalies
  - Reduce the pressure to have null values
- Normalization puts relations in good form!
- Normalization is a solved problem *(all algorithms & proofs are worked out)*

Decomposing to Normalize

- There are three potential problems to consider:
  - Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation! *(Losslessness)*
  - Checking some dependencies may require joining the instances of the decomposed relations. *(Dependency preservation)*
  - Some queries become more expensive.
    - e.g., In which project does John work? (EMP2 JOIN X)
- *Tradeoff*: Must consider these issues vs. redundancy.

How do we know if a decomposition is correct? That we haven't lost anything?

We have three goals:

- Lossless decomposition
  - (don't throw any information away)
  - (be able to reconstruct the original relation)
- Dependency preservation
  - all of the non-trivial FDs each end up in just one relation (not split across two or more relations)
- Boyce-Codd normal form (BCNF)
  - no redundancy beyond foreign keys -- all FDs implied by keys

What is a lossless decomposition?

When R is decomposed into R_1 and R_2

If $(R_1 \bowtie R_2) = R$ then decomposition is lossless

if it is a lossy decomposition, then $R_1 \bowtie R_2$ gives you TOO MANY tuples.

Test for a Lossless Decomposition

Let R_1 and R_2 form a decomposition of R.
R_1 and R_2 are both sets of attributes from R.

The decomposition is lossless if ...

the attributes in common are a key for *at least one* of the relations R_1 and R_2

$$R_1 \cap R_2 = \text{key}(R_1), \text{ or}$$
$$R_1 \cap R_2 = \text{key}(R_2)$$

Example: testing for a lossless decomposition

Employee(<u>SS-number</u>, name, project, p-title)

decomposition: Employee (<u>SS-number</u>, name)
Project (<u>project</u>, p-title, <u>name</u>)

Which attribute is in common?

name (of the employee)

Is name a key for either of these two tables?

NO! We have a problem.

Given a set of decomposed relations, use the Chase Test to determine whether the decomposition is lossless!

Closure of Attributes

Given a relation R(A1,…,An) and a set of FDs F. The closure of (A1,…,An) under F, denoted by (A1, …,An)$^+$, is the set of attributes B such that every relation that satisfies F, also satisfies

$$A1,\ldots,An \rightarrow B$$

In other words, A1,…,An → B *follows* from F

Computing Closure of Attributes: Example

R(A,B,C,D,E,F)
FDs F = {AB→C, BC→AD, D→E, CF→B}

Is AB a key for R?

What is the closure of:

$\{A,B\}^+ = \{A, B\}$ *trivial dependencies*
$= \{A,B,C\}$ *AB→C*
$= \{A,B,C,D\}$ *BC → D*
$= \{A,B,C,D,E\}$ *D → E*

Now, we know that AB → CDE
F ==> AB → CDE, or AB → CDE follows from F

Closure of FDs

- Let F be a set of FDs

F^+ -- the closure of F, is the set of all FDs implied from F

- Closure can be computed by:
  – using a set of *inference rules:* apply rules until no new FDs arise -- until a *fixpoint* is reached
  – Use algorithm for closing a set of attributes: close all possible subsets!

Normal Forms

- Returning to the issue of schema refinement, the first question to ask is whether any refinement is needed!
- If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized.
  - This can be used to help us decide whether decomposing the relation will help.

Normal Forms Based on FDs

1NF – all attribute values (domain values) are atomic
(part of the definition of the relational model)

2NF – all non-key attributes must depend on a **whole** candidate key (no partial dependencies)

3NF – table is in 2NF and all non-key attributes must depend on **only** a candidate key (no transitive dependencies)

BCNF – every determinant is a superkey, X --> A, X is a superkey

BCNF >> 3NF >> 2NF >> 1NF

skip

Normalization Made Easy (Ronald Fagin)

Every attribute must depend
upon the key, ← definition of key
the whole key, ← 2NF
and
nothing but the key. ← BCNF (and 3NF)

Examples of Violations

2NF - all *non-key* attributes must depend on the **whole** key

Assigned-to (<u>A-project, A-emp,</u> emp-name, percent)

3NF - all *non-key* attributes must depend on **only** the key

Employee (<u>SS-number</u>, name, address, project, p-title)

BCNF - every determinant is a candidate key

(all FDs are implied by the candidate keys)

Assigned-to (<u>A-emp-ID, A-Project,</u> A-SS-number, percent)

What's the Goal?

BCNF and Lossless and Dependency-Preserving

(first choice)

3NF and Lossless and Dependency-Preserving

(second choice)

because sometimes we can't preserve all dependencies

Dependency-Preservation: Example

R = addr(city, state, zip)

FDs = city state → zip, zip → state

Decomposition: R1(zip,state), R2(city,zip)

city state → zip *does not hold in R2*

Problem: If the DBMS only enforces keys (and not FDs directly), this FD won't be enforced

Testing *city state → zip* requires a join: R1 JOIN R2

What is "Dependency Preserving"

Suppose F is the original set of FDs and G is set of *projected* FDs after decomposition

The decomposition is dependency preserving if $F^+ \equiv G^+$. That is, the two closures must be equivalent.

Projecting Functional Dependencies

- To check for dependency preservation we need to determine which FDs hold for the decomposed relations—we can do this by *projecting* the dependencies
- Given a relation R with FDs F. Let S = π_A R
- What FDs hold in S?
  - All FDs *f* that follow from F, i.e., $f \in F^+$, that involve only attributes of S

Projecting FDs: Example

R(A,B,C,D)

FDs: $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow D$

Which FDs hold for S(A,C,D) ?

{A}+ = {A,B,C,D}, thus $A \rightarrow C$ and $A \rightarrow D$ hold in S

  $A \rightarrow B$ makes no sense in S!

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

  B→anything makes no sense in S!

{C}+ = {C,D}, thus $C \rightarrow D$ hold in S

{D}+ = {D}, thus $D \rightarrow D$ hold in S

  trivial dependency

Projecting FDs: Example (cont.)

R(A,B,C,D)

FDs: A→B, B→C, and C→D

Which FDs hold for S(A,C,D) ? *A→C, A→D, C→D*

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

{A anything} will add nothing new – everything will *follow* by augmentation

{CD}+ = {C,D}

nothing new

Stop!

Redundancy in FDs and Minimal Covers

- $A \rightarrow B$, $B \rightarrow C$, $A \rightarrow C$
  - $A \rightarrow C$ is redundant – it can be inferred from $A \rightarrow B$, $B \rightarrow C$
- A *minimal cover* (or basis) for a set F of FDs is a set G of FDs s.t.:
  1. Every dependency in G is of the form $X \rightarrow A$ and A is a single attribute
  2. F+ = G+
  3. If H = (G – $X \rightarrow A$), then H+ <> F+

Every dependency is as small as possible, and required for the closure to be equal to F+

Computing Minimal Covers

- Given a set of FDs F:

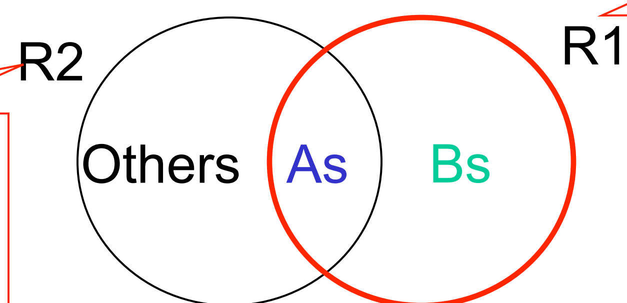
1. Put FDs in standard form
   - Obtain G of equivalent FDs with single attribute on the right side
2. Minimize left side of each FD
   - For each FD in G, check each attribute on the left side to see if it can be deleted while preserving equivalence to F+
3. Delete redundant FDs
   - Check each remaining FD in G if it can be deleted while preserving equivalence to F+

Decomposition and Normal Forms

Decomposition into BCNF

- We can break *any* relation schema R into a collection of subsets Si of its attributes s.t.:
  - Si is in BCNF
  - We don't lose information, i.e., we can *reconstruct* R from Si
- Suppose A1, A2, ..., An → B1,…,Bm violates BCNF. Construct 2 overlapping relations R1 and R2:
  - R1 = A U B
  - R2 = (R - (B-A))

All attributes involved in the violation



Left side + all attributes not involved in the FD

BCNF: Example

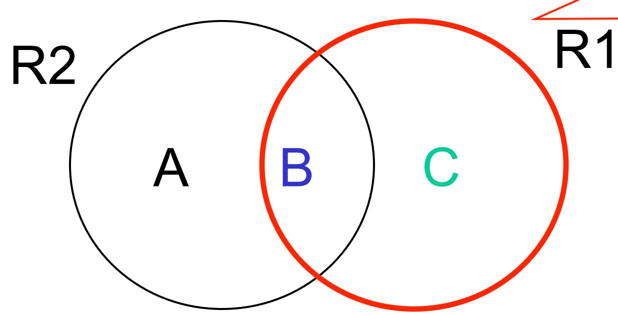
- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $B \rightarrow C\}$ – violates BCNF

Key = $\{A\}$

- R is not in BCNF – B is not a superkey
- Decomposition $R_1 = (B, C)$, $R_2 = (A, B)$,

R_1 and R_2 in BCNF

All attributes involved in the violation



Left side + all attributes not involved in the FD

Testing for BCNF

- To check if a non-trivial dependency $A \rightarrow B$ causes a violation of BCNF
  1. Compute A^+
  2. Verify that A is a superkey of *R*.

- To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F*
  - If none of the dependencies in *F* causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.

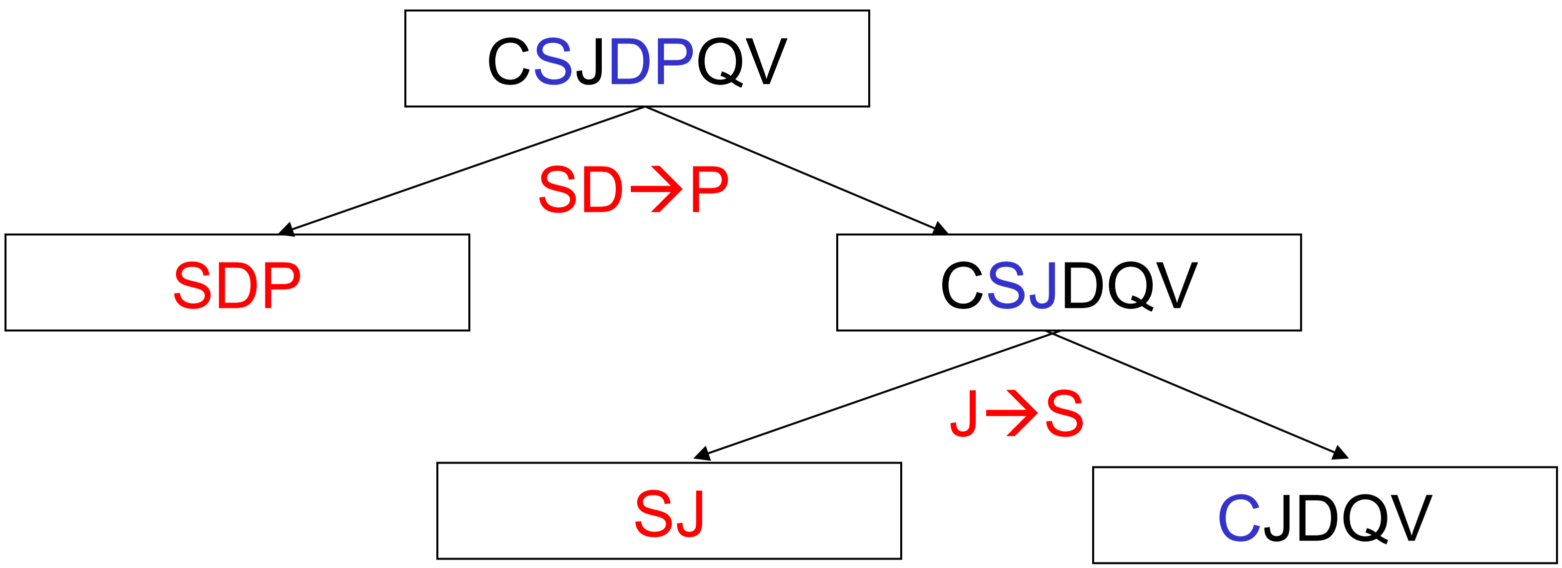
- However, using only F is incorrect when testing a relation in a decomposition of R

Testing for BCNF: Example

- Consider $R(A, B, C, D)$, with $F = \{ A \rightarrow B, B \rightarrow C\}$
- Decompose R into $R_1(A,B)$ and $R_2(A,C,D)$
  - R1 is in BCNF. *Why?*
    - *$A \rightarrow B$ holds in R_1 and $A+ = AB$*
  - Since neither of the dependencies in F contain only attributes from (A,C,D) , *does this mean R_2 is in BCNF?*
  - $A \rightarrow C$ in F^+ holds in R_2 and A is not a superkey for R_2
  - *R_2 is NOT in BCNF!*

Decomposing into BCNF: Example

- R = CSJDPQV
- F = {SD→P; J→S; JP→C}, Key = {C}

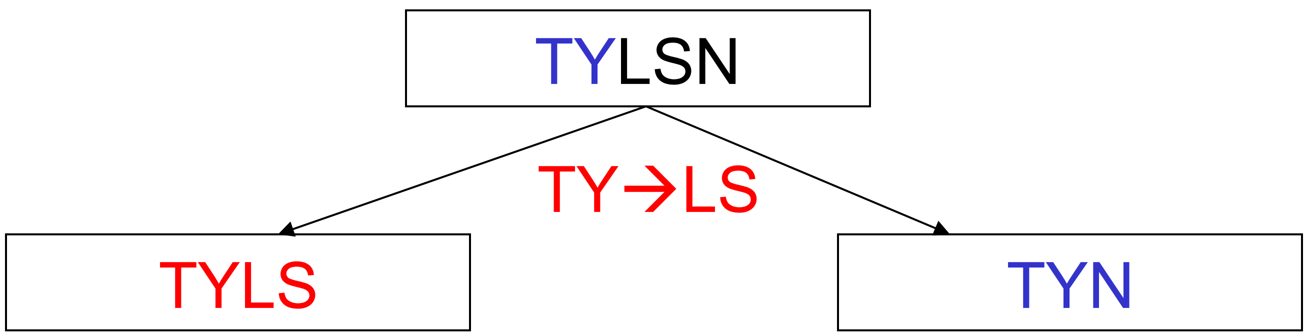


Need to check for BCNF at each step!
Note: any 2-attribute relation is in BCNF (see pg 89 in textbook)

Decomposing into BCNF: Another Example

- Movies = {title,year,length,studio,starName}
- F = {title,year→length,studio},

Key = {title,year,starName}



Movies = {title,year,length,studio}

StarsIn = {title,year,starName}

3NF: The next best to BCNF

- Recall that: A relation schema R is in third normal form (3NF) if for all:

  $$A \rightarrow B \text{ in } F^+$$

  at least one of the following holds:
  - $A \rightarrow B$ is trivial (i.e., $B \subseteq A$)
  - A is a superkey for R
  - Each attribute t in $B - A$ is contained in a candidate key for R

$\left.\begin{array}{l}\text{first two conditions}\end{array}\right\}$ BCNF

relaxation of BCNF to ensure dependency preservation

Any relation can be decomposed into 3NF in a dependency-preserving manner

R = addr(city, state, zip)

FDs = city state → zip, zip → state

R is in 3NF

(city state) is a superkey

(state) is contained in a key *%% state-zip*

R preserves all FDs but …

there is some redundancy in this schema

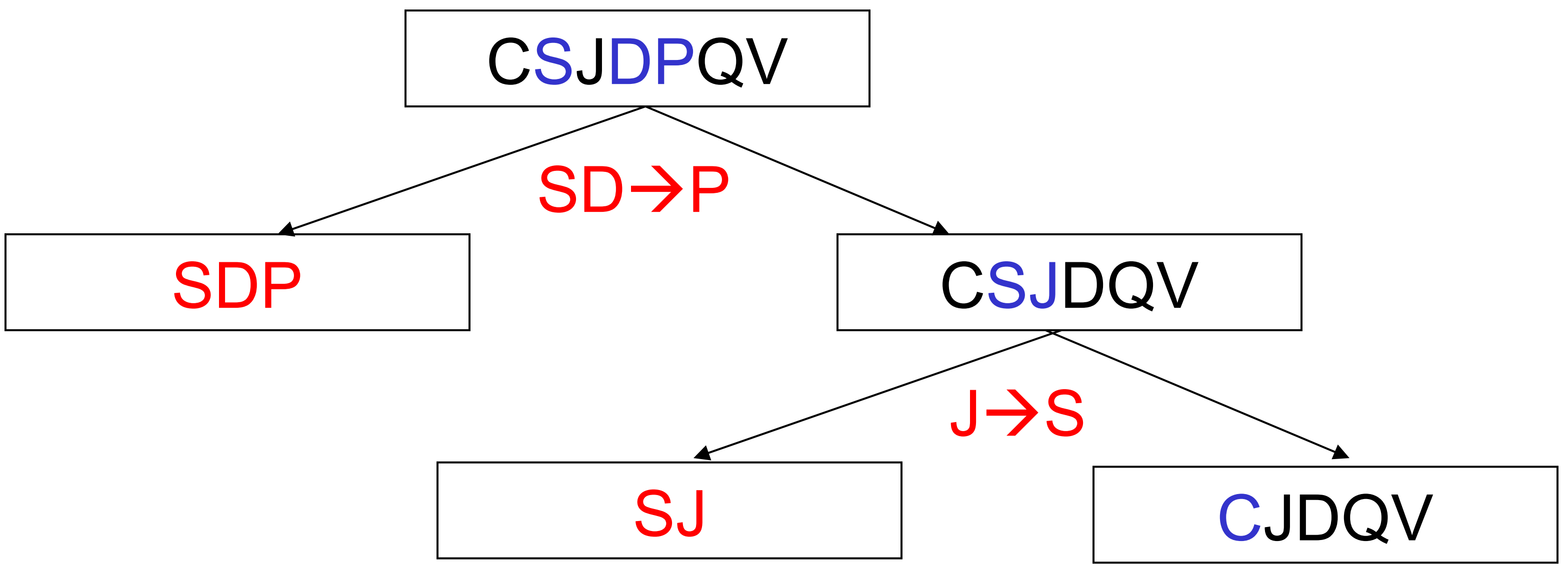
Decomposition into 3NF

- Apply BCNF decomposition
- Identify the set N of dependencies that is not preserved
- For each X→A in N, add relation XA to schema

- Note: need to work with the *minimal basis* of the functional dependencies!

Dependency-Preserving Decomposition into 3NF: Example

- R = CSJDPQV
- F = {SD→P; J→S; JP→C}, Key = {C}



JP→C is not preserved

Since F is a minimal cover, add CJP to the schema

Semi-Structured Data and XML

Bridging the Gap: Data on the Web vs. Data in Databases

XML: Documents and Data

- A well-formed document is a XML document that follows the basic rules:
  - single root element,
  - matched tags,
  - unique attribute names, etc.
- A document is valid wrt a schema
  - For XML, schemas are descriptive rather than prescriptive

XML Schema

- Defines:
  - vocabulary (element and attribute names)
  - content model (relationships and structure)—regular expressions
  - data types
- Written in XML
- Often uses namespace abbreviated as xs or xsd
- Namespace declaration:

  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

XML Schema Types

- Simple and complex element types

  Simple: <shipDate>2007-10-16</shipDate>

  Complex:

  ```
  <purchaseOrder orderDate="2007-10-15">
    <shipTo>…</shipTo>
    …
  </purchaseOrder>
  ```

- An element with attributes is always complex
- Attributes are unordered
- Can *restrict* attribute or element values

Number of Occurrences

- Number of times an element appears in a document: minOccurs and maxOccurs
- Default values:
  - minOccurs: 1
  - maxOccurs: 1
- <xsd:element name="comment" minOccurs="0"/>
- <xsd:element name="item" minOccurs="0"
  maxOccurs="unbounded"/>
- maxOccurs can be *unbounded,* allowing an unlimited number of those elements

Common Querying Tasks

- Filter, select XML values
  - Navigation, selection, extraction
- Merge, integrate values from multiple XML sources
  - Joins, aggregation
- *Transform XML values from one schema to another*
  - *XML construction*

- Programmatic interfaces (DOM/SAX) specify how
- Query languages specify what, not how
  - Provide abstractions for common tasks
  - Easier than programmatic interfaces

Query Languages

- XPath 2.0
  - Common language for navigation, selection, extraction
  - Used in XSLT, XQuery, XPointer, XML Schema, XForms, et al
- XSLT 2.0: XML ⇒ XML, HTML, Text
  - Loosely-typed scripting language
  - Format XML in HTML for display in browser
  - Must be highly tolerant of variability/errors in data
- XQuery 1.0: XML ⇒ XML
  - Strongly-typed query language
  - Large-scale database access
  - Must guarantee safety/correctness of operations on data
- Over time, XSLT & XQuery may both serve needs of many application domains

XPath

- In its simplest form, an XPath is like a path in a file system:

  /mypath/subpath/*/morepath

  - The XPath returns a *node set* representing the XML nodes (and their subtrees) at the end of the path
  - XPaths can have *node tests* at the end, returning only particular node types, e.g., text(), processing-instruction(), comment(), element(), attribute()
  - XPath is fundamentally an ordered language: it can query in order-aware fashion, and it returns nodes in order

XPath

- **XPath = sequence of location steps**
- A *location step* is:
  axis-name::*node-test*[*predicate*]
- Example: descendant::book[@title="XML"]
- axes: self, child, parent, descendant, ancestor, descendant-or-self, ancestor-or-self, following, preceding, following-sibling, preceding-sibling
- Steps are joined by forward slashes
- Example: root()/child::imdb/descendant-or-self::node()/child::title
- Many syntax shortcuts: /imdb//title

XPath

- Syntax for navigating XML
- Looks similar to file paths
- Used by XML Schema, XSLT, XQuery
- Searches by structure and text
- Guarantees same syntactic expression has same semantics
- Navigation, selection, value extraction
- Arithmetic, logical, comparison expressions

XQuery FLWOR

- SQL:

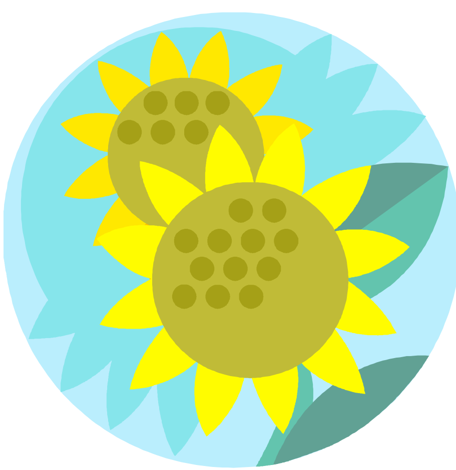
  SELECT <attribute list>

  FROM <set of tables>

  WHERE <set of conditions>

  ORDER BY <attribute list>

- XQuery: **F**OR-**L**ET-**W**HERE-**O**RDERBY-**R**ETURN



FOR/LET Clauses

↓ List of tuples

WHERE Clause

↓ List of tuples

ORDERBY/RETURN Clause

Instance of XQuery data model

XQuery: Example

For each actor, return box office receipts of films in which they starred in past 2 years

```
let $imdb := document("www.imdb.com/imdb.xml")
for $actor in $imdb//actor
let $films :=
  $imdb//show[box_office and @year >= 2000
    and $actor/name = .//actor[@role="star"]/name]
return
 <receipts>
   { $actor }
   <total> { sum($films/box_office) } </total>
 </receipts>
```

Iteration

Join

XML Construction

Aggregation

XQuery

- FOR $x in expr -- binds $x to each value in the list expr
- LET $x := expr -- binds $x to the entire list expr
  - Useful for common subexpressions and for aggregations

FOR vs. LET

Returns:

<result> <show>...</show></result>
<result> <show>...</show></result>
<result> <show>...</show></result>
...

```
FOR $x IN document("imdb.xml")//show
RETURN <result> $x </result>
```

```
LET $x := document("imdb.xml")//show
RETURN <result> $x </result>
```

Returns:

<result> <show>...</show>
<show>...</show>
<show>...</show>
...
</result>

Aggregates

Find movies whose box office proceeds are larger than average:

```
LET $a := avg(document("imdb.xml")//box_office)
FOR $s in document("imdb.xml")//show
WHERE $s//box_office > $a
RETURN $s
```

Challenge Question

Are the following queries equivalent?

A.
```
FOR $show IN document("www.imdb.com/imdb.xml")//show,
 $review IN $show/review
WHERE
 $show/@year >= 2002
RETURN
<show> <t>$show/title</t> <r>$review</r> </show>
```

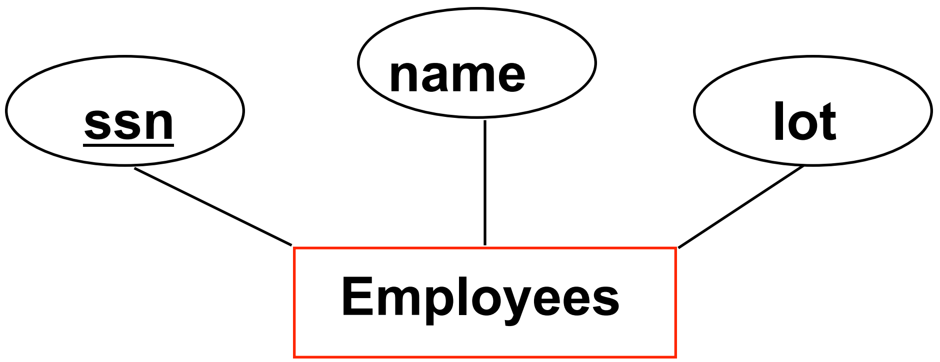
B.
```
FOR $show IN document("www.imdb.com/imdb.xml")//show
 WHERE
 $show/@year >= 2002
 RETURN
 <show> <t>$show/title</t> <r>$show/review</r> </show>
```

Conceptual Design using the Entity Relationship (ER) Model

ER Model

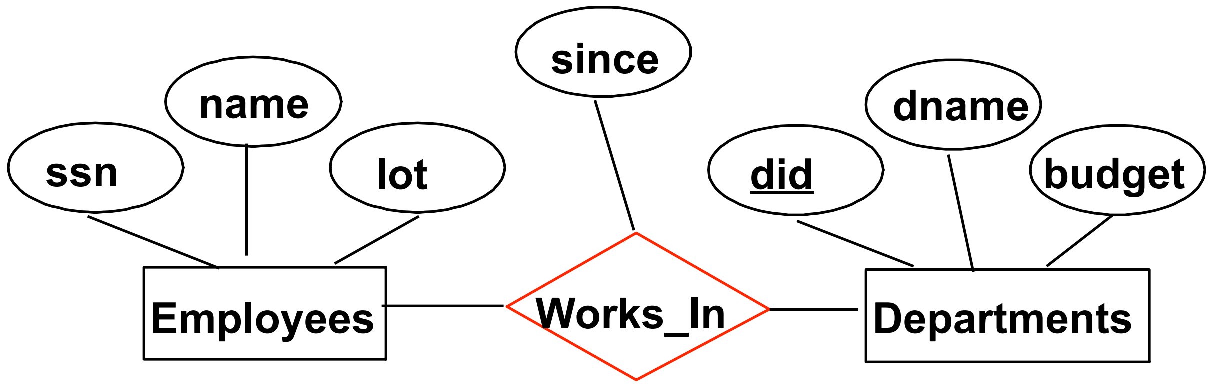
- *Provides a language to design a conceptual model of a database*
  - What are the *entities* and *relationships* among these entities in the enterprise?
  - What information about these entities and relationships should we store in the database?
  - What are the *integrity constraints* or *business rules* that hold?
  - A database `schema' (structure) in the ER Model can be represented pictorially (*ER diagrams*).
- Can map an ER diagram into a relational schema.

ER Model - Entities



- Entity: Real-world object distinguishable from other objects
  - E.g., specific person, company, event
  - described (in DB) using a set of *attributes*
  - Values for a set of attributes uniquely identify entity
- Entity Set: A collection of similar entities that share the same properties (attributes)
  - E.g., all employees, set of all persons, companies, events

ER Model - Relationships

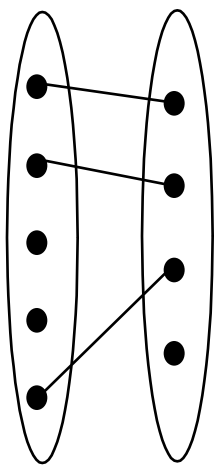


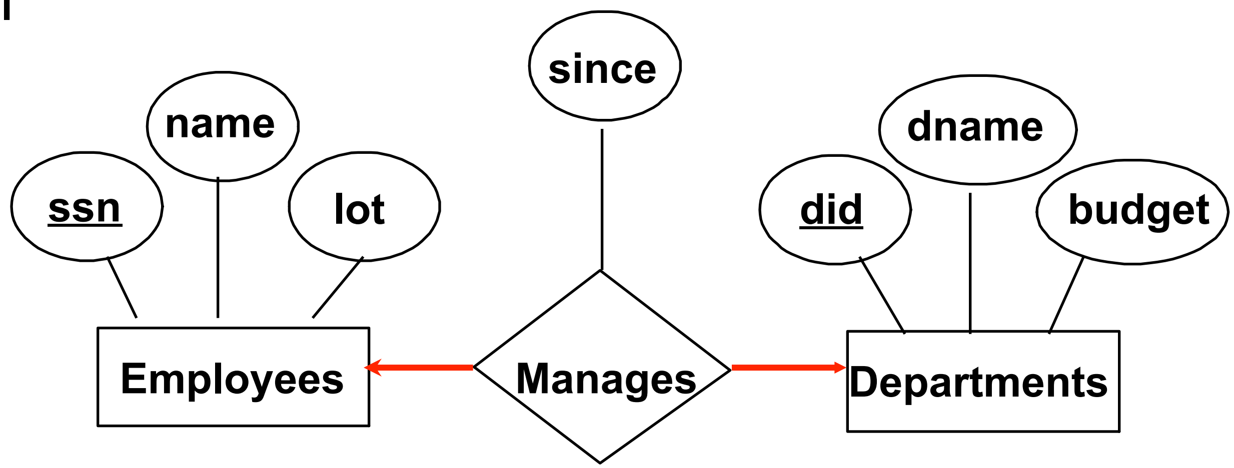
- Relationship: connection among two or more entity sets
  – E.g., the employee John works in Pharmacy department
- To create an instance of a relationship, we must indicate which employee and which department we want to have connected (for this relationship).
- We need the key value for an employee and the key value for a department, stored together, to represent the relationship

Multiplicity of Relationships

How many times must/may an entity instance participate?

- Each dept has *at most* one manager, and a manager can manage *at most* one department

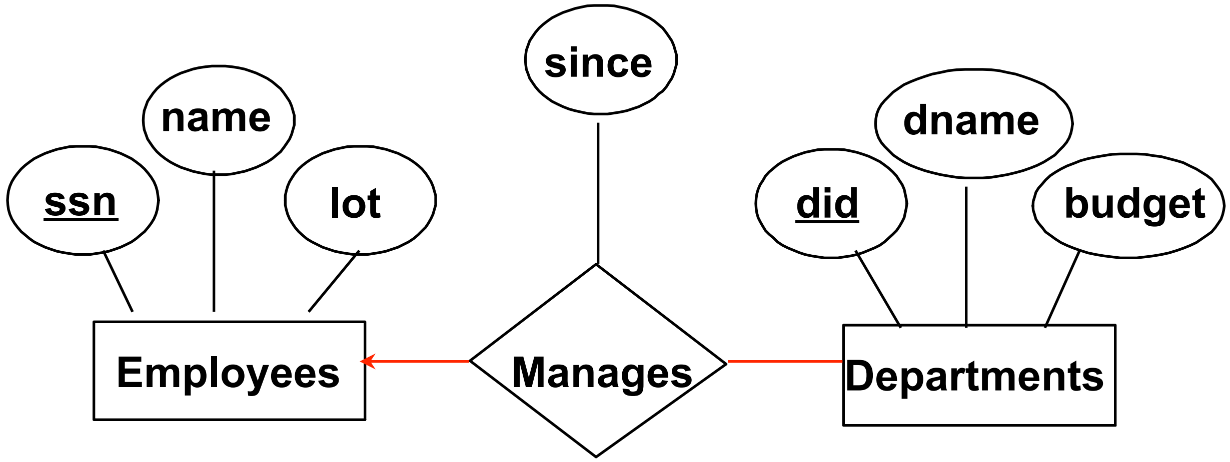


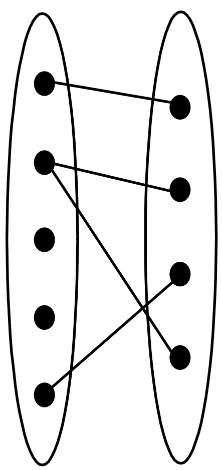


One-to-one relationship

Multiplicity of Relationships

- Each dept has *at most* one manager, but an employee can manage multiple departments





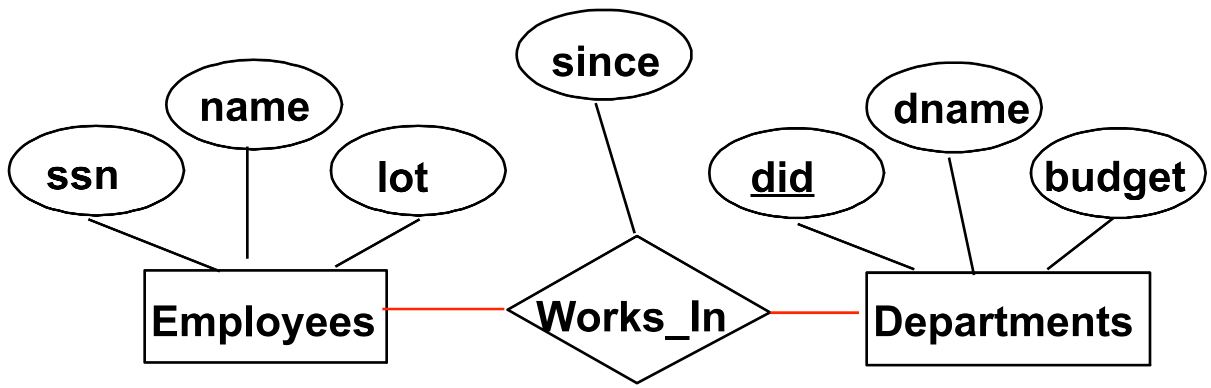
1-to Many

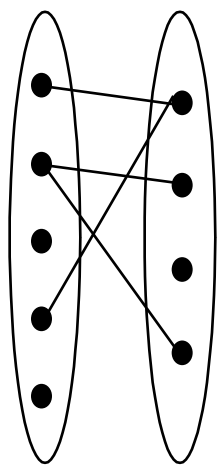
One-to-many relationship
(employees→departments)
Many-to-one relationship
(departments→employees)

Multiplicity of Relationships

How many times must/may an entity instance participate?

- An employee can work in many departments; a dept can have many employees.



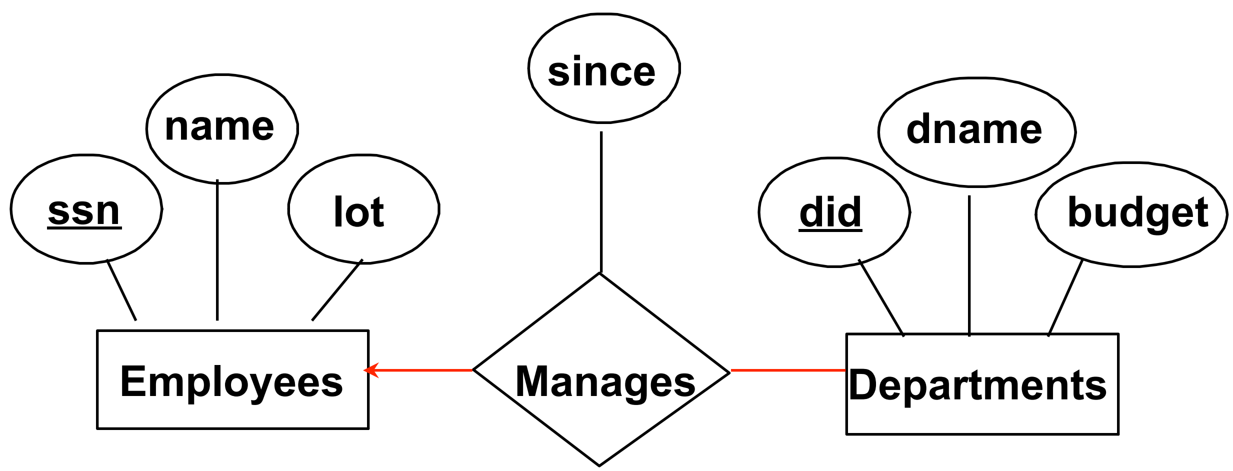


Many-to-Many

Many-to-many relationship

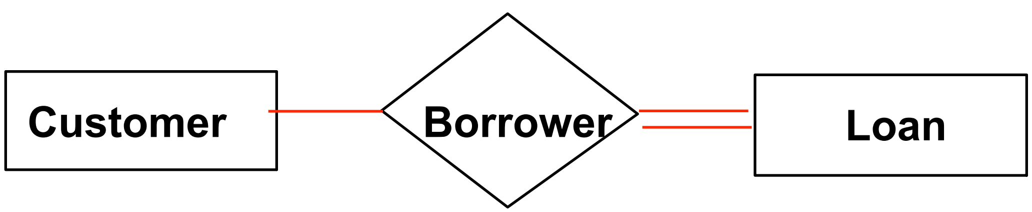
Semantics of the "arrow"

- Arrow means *at most one*
- It does not guarantee existence of an entity set pointed to
- E.g., there can be departments without a manager at a particular point in time



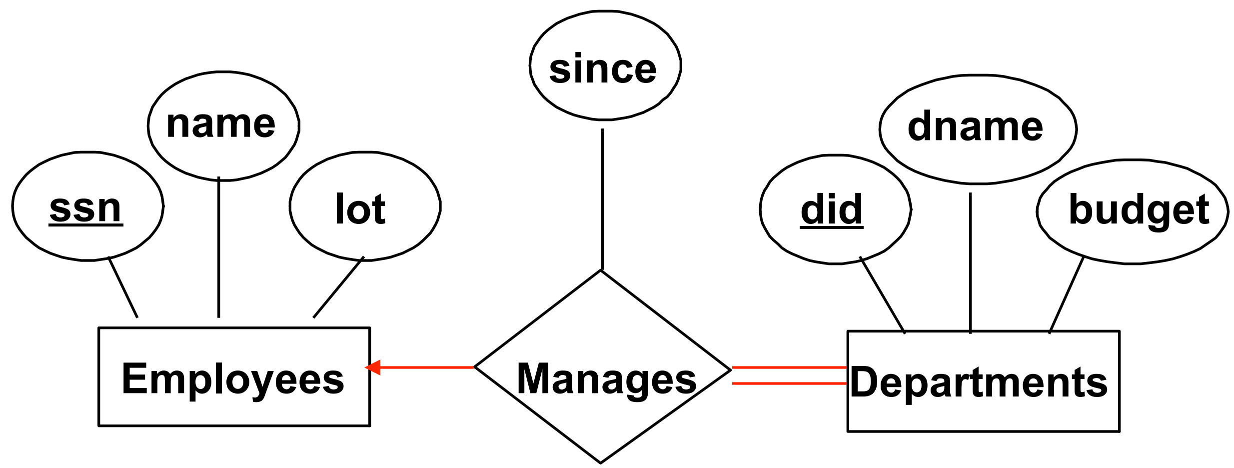
Participation Constraints

- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
  - participation of *loan* in *borrower* is total -- every loan must have a customer associated to it via borrower; a *loan* cannot exist without being in at least one *borrower* relationship
- Partial participation: some entities may not participate in any relationship in the relationship set
  - participation of *customer* in *borrower* is partial, some customers only have a checking account



Referential Integrity

- The many-to-one relationship Manages states that a department has *at most* one manager, it may have no manager
- Combined with total participation asserts that *exactly one* value exists for a particular role



Forbid the deletion of an employee that manages a department, or delete both the employee and department!

Additional Constructs

- ISA relationships
- Weak entity sets

Translating an ER Diagram into a Relational Schema

First Approximation

- Turn each entity into a relation with the same set of attributes
- Replace relationship by a relation whose attributes are the keys for the connected entity sets
- Deal with *special situations*
  - Multi-valued and composite attributes
  - Weak entity sets
  - ISA relationships
  - Relations that result from an entity set and 1-to-many relationship