CMSC 461, Database Management Systems
Spring 2018

Lecture 13 - Chapter 8 Relational Database Design Part 3

These slides are based on “Database System Concepts” 6th edition book and are a modified version of the slides which accompany the book (http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/index.html), in addition to the 2009/2012 CMSC 461 slides by Dr. Kalpakis

Dr. Jennifer Sleeman

Logistics

- Midterm 3/14/2018

Lecture Outline

- ***Functional Dependencies Review***
- BCNF Decomposition
- Database Design Process
- Midterm Review

Functional Dependencies

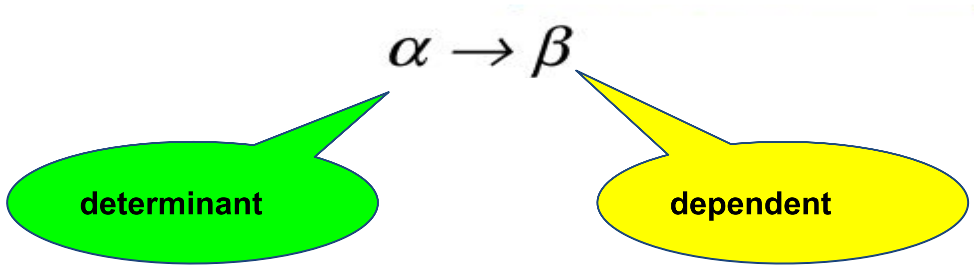
We represent constraints by using keys

- super keys
- primary keys
- candidate keys

We can also represent constraints using functional dependencies

Functional Dependencies

- *A functional dependency* is a relationship between two attributes, where ***β*** *is functionally dependent on* ***α***
- ***α*** *is usually the primary key*
- For every valid instance of ***α***, that value uniquely determines the value of ***β***.



Functional Dependencies

What are the dependencies in Table Foo?

Table Foo:				
A	B	C	D	E
a1	b1	c1	d1	e1
a2	b1	c2	d2	e1
a3	b2	c1	d1	e1
a4	b2	c2	d2	e1

Based on and image from "Database System Concepts" book and slides, 6th edition, example table from https://opentextbc.ca/dbdesign01/chapter/chapter-11-functional-dependencies/

Functional Dependencies

Can we say this:

$A \rightarrow B, \quad A \rightarrow C, \quad A \rightarrow D, \quad A \rightarrow E$

Table Foo:				
A	B	C	D	E
a1	b1	c1	d1	e1
a2	b1	c2	d2	e1
a3	b2	c1	d1	e1
a4	b2	c2	d2	e1

Functional Dependencies

If we can say this:

$A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $A \rightarrow E$

Then we can say:

$A \rightarrow BC$ (or any other subset of ABCDE)

Table Foo:				
A	B	C	D	E
a1	b1	c1	d1	e1
a2	b1	c2	d2	e1
a3	b2	c1	d1	e1
a4	b2	c2	d2	e1

Based on and image from "Database System Concepts" book and slides, 6th edition, example table from https://opentextbc.ca/dbdesign01/chapter/chapter-11-functional-dependencies/

Functional Dependencies

We can summarize this as
$A \rightarrow BCDE$

Table Foo:				
A	B	C	D	E
a1	b1	c1	d1	e1
a2	b1	c2	d2	e1
a3	b2	c1	d1	e1
a4	b2	c2	d2	e1

Based on and image from "Database System Concepts" book and slides, 6th edition, example table from https://opentextbc.ca/dbdesign01/chapter/chapter-11-functional-dependencies/

Functional Dependencies

Other dependencies we can observe?

Table Foo:

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b1	c2	d2	e1
a3	b2	c1	d1	e1
a4	b2	c2	d2	e1

Functional Dependency Theory

We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies.

Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by *F*
  - For example:
    Given a schema r(A,B,C)
    If $A \rightarrow B$ and $B \rightarrow C$
    then we can infer that $A \rightarrow C$
- The set of all functional dependencies logically implied by *F* is the closure of *F*
- We denote the closure of *F* by F^+
- F^+ is a superset of *F*

Closure of a Set of Functional Dependencies

- We can find F^+, the closure of F, by repeatedly applying **Armstrong's Axioms:**
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (**reflexivity**)
  - if $\alpha \rightarrow \beta$, then $\gamma\,\alpha \rightarrow \gamma\,\beta$ (**augmentation**)
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (**transitivity**)
- These rules are
  - **sound** (generate only functional dependencies that actually hold), and
  - **complete** (generate all functional dependencies that hold).

Computing F+

- To compute the closure of a set of functional dependencies F:

```
  $F^+ = F$
repeat
for each functional dependency $f$ in $F^+$
      apply reflexivity and augmentation rules on $f$
      add the resulting functional dependencies to $F^+$
for each pair of functional dependencies $f_1$ and $f_2$ in $F^+$
      if $f_1$ and $f_2$ can be combined using transitivity
 then add the resulting functional dependency to $F^+$
until $F^+$ does not change any further
```

Closure of a set of Functional Dependencies

- Additional rules:
  - If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (**union**)
  - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (**decomposition**)
  - If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds (**pseudotransitivity**)

The above rules can be inferred from Armstrong's axioms.

Closure of a set of Functional Dependencies Example

- $R = (A, B, C, G, H, I)$

  $F = \{ A \rightarrow B$

  $A \rightarrow C$

  $CG \rightarrow H$

  $CG \rightarrow I$

  $B \rightarrow H\}$
- some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ and $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - *by union rule, since $CG \rightarrow H$ and $CG \rightarrow I$, implies $CG \rightarrow HI$*

Closure of Attribute Sets

- Given a set of attributes α, define the ***closure*** of α **under** F (denoted by α^+) as the set of attributes that are functionally determined by α under F

- Algorithm to compute α^+, the closure of α under F

```
result := $\alpha$;
while (changes to result) do
    for each $\beta \rightarrow \gamma$ in $F$ do
    begin
    if $\beta \subseteq result$ then  $result := result \cup \gamma$
    end
```

Closure of Attribute Sets Example

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. $result = ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$

Closure of Attribute Sets Uses

There are several uses of the attribute closure algorithm:

- Testing for superkey:
  - To test if α is a superkey, we compute α^{+}, and check if α^{+} contains all attributes of R.
- Testing functional dependencies
  - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^{+}), just check if $\beta \subseteq \alpha^{+}$.
  - That is, we compute α^{+} by using attribute closure, and then check if it contains β.
  - Is a simple and cheap test, and very useful
- Computing closure of F
  - For each $\gamma \subseteq R$, we find the closure γ^{+}, and for each $S \subseteq \gamma^{+}$, we output a functional dependency $\gamma \rightarrow S$.

Closure of Attribute Sets Examples

- $R = (name, color, category, department, price)$
- $F = \{$name $\rightarrow$ color
  $category \rightarrow$ department
  $color, category \rightarrow$ price}

You find:

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

Lecture Outline

- Functional Dependencies Review
- ***BCNF Decomposition***
- Database Design Process
- Midterm Review

Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

  $$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C)$
  - Can be decomposed in two different ways
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless-join decomposition:
    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
  - Dependency preserving
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Lossless-join decomposition:
    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i.
  - A decomposition is **dependency preserving**, if $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
  - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Testing for Dependency Preservation

- To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of R into $R_1, R_2, \ldots, R_n$ we apply the following test (with attribute closure done with respect to F)
  -
    ```
    result = α
    while (changes to result) do
     for each R_i in the decomposition
        t = (result ∩ R_i)^+ ∩ R_i
        result = result ∪ t
    ```
  - If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved.

Testing for Dependency Preservation

- We apply the test on all dependencies in *F* to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute F^+ and $(F_1 \cup F_2 \cup \ldots \cup F_n)^+$

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $B \rightarrow C\}$
  Key = $\{A\}$
- R is not in BCNF
- Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$
  - R_1 and R_2 in BCNF
  - Lossless-join decomposition
  - Dependency preserving

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
  1. compute α^+ (the attribute closure of α), and
  2. verify that it includes all attributes of *R*, that is, it is a superkey of *R*.

Testing for BCNF

- **Simplified test**: To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F* for violation of BCNF, rather than checking all dependencies in 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.

Testing for BCNF

- However, **simplified test using only *F* is incorrect when testing a relation in a decomposition of R**
  - Consider $R = (A, B, C, D, E)$, with $F = \{ A \rightarrow B, BC \rightarrow D\}$
    - Decompose R into $R_1 = (A,B)$ and $R_2 = (A,C,D, E)$
    - Neither of the dependencies in F contain only attributes from (A,C,D,E) so we might be mislead into thinking R_2 satisfies BCNF.
    - In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.

Testing Decomposition for BCNF

- To check if a relation R_i in a decomposition of R is in BCNF,
  - Either test R_i for BCNF with respect to the **restriction** of F to R_i (that is, all FDs in F^+ that contain only attributes from R_i)
  - or use the original set of dependencies F that hold on R, but with the following test:
    - for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.
  - If the condition is violated by some $\alpha \rightarrow \beta$ in F, the dependency

    $$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$

    can be shown to hold on R_i, and R_i violates BCNF.
  - We use above dependency to decompose R_i

Example of BCNF Decomposition

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $B \rightarrow C\}$
  Key = $\{A\}$
- R is not in BCNF ($B \rightarrow C$ but B is not superkey)
- Decomposition
  - $R_1 = (B, C)$
  - $R_2 = (A,B)$

Example of BCNF Decomposition

- *class* (*course_id*, *title*, *dept_name*, *credits*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)
- Functional dependencies:
  - *course_id*→ *title*, *dept_name*, *credits*
  - *building*, *room_number*→*capacity*
  - *course_id*, *sec_id*, *semester*, *year*→*building*, *room_number*, *time_slot_id*

Example of BCNF Decomposition

- A candidate key {*course_id*, *sec_id*, *semester*, *year*}.
- BCNF Decomposition:
  - *course_id*→ *title*, *dept_name*, *credits* holds
    - but *course_id* is not a superkey.
  - We replace *class* by:
    - *course*(*course_id*, *title*, *dept_name*, *credits*)
    - *class-1* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number, capacity*, *time_slot_id*)

BCNF Decomposition

- *course* is in BCNF
  - How do we know this?
- *building*, *room_number*→*capacity* holds on *class-1*
  - but {*building*, *room_number*} is not a superkey for *class-1*.
  - We replace *class-1* by:
    - *classroom* (*building*, *room_number*, *capacity*)
    - *section* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
- *classroom* and *section* are in BCNF.

BCNF and Dependency Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving

- $R = (J, K, L)$
  $F = \{JK \rightarrow L$
  $L \rightarrow K\}$
  Two candidate keys = *JK* and *JL*
- *R* is not in BCNF
- Any decomposition of *R* will fail to preserve
  $JK \rightarrow L$
  This implies that testing for $JK \rightarrow L$ requires a join

Design Goals

- Goal for a relational database design is:
  - BCNF.
  - Lossless join.
  - Dependency preservation.
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation
  - Redundancy due to use of 3NF

Design Goals

- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.
  Can specify FDs using assertions, but they are expensive to test, (and currently not supported by any of the widely used databases!)
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

Lecture Outline

- Functional Dependencies Review
- BCNF Decomposition
- ***Database Design Process***
- Midterm Review

Overall Database Design Process

- We have assumed schema *R* is given
  - *R* could have been generated when converting E-R diagram to a set of tables.
  - *R* could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
  - Normalization breaks *R* into smaller relations.
  - *R* could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
  - Example: an *employee* entity with attributes
    department_name and *building*,
    and a functional dependency
    department_name → *building*
  - Good design would have made department an entity
- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary

Denormalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying *prereqs* along with *course_id,* and *title* requires join of *course* with *prereq*
- Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes
  - faster lookup
  - extra space and extra execution time for updates
  - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined as

  $course \bowtie prereq$
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Other Design Issues

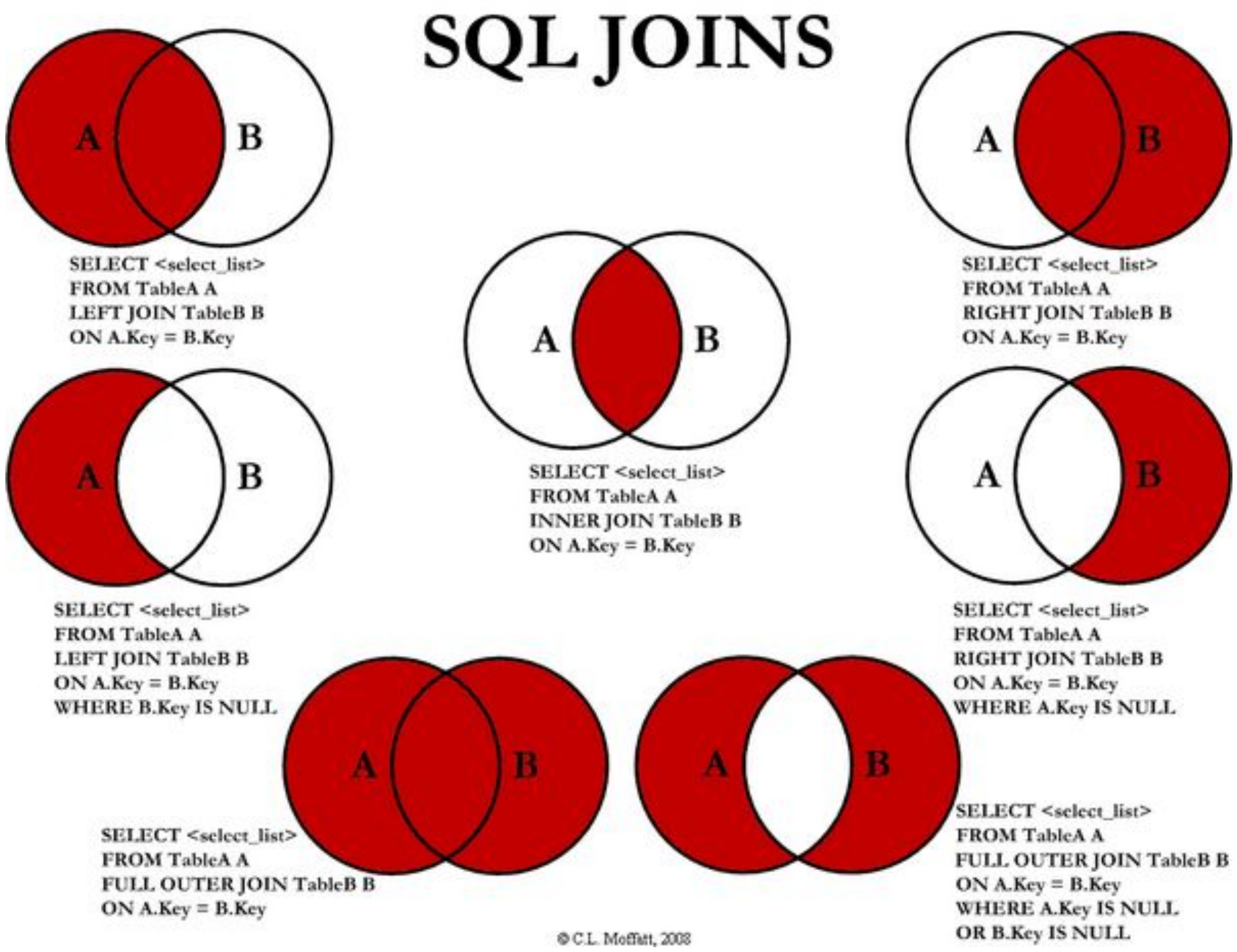
- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:
  Instead of *earnings* (*company_id, year, amount*), use
  - *earnings_2004, earnings_2005, earnings_2006*, etc., all on the schema (*company_id, earnings*).
    - Above are in BCNF, but make querying across years difficult and needs new table each year
  - *company_year* (*company_id, earnings_2004, earnings_2005,*
    earnings_2006)
    - Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
    - Is an example of a **crosstab**, where values for one attribute become column names
    - Used in spreadsheets, and in data analysis tools

Lecture Outline

- Functional Dependencies Review
- BCNF Decomposition
- 3NF Decomposition
- Multivalued Decomposition
- Fourth Normal Form
- Database Design Process
- ***Midterm Review***

Midterm Review

Joins



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Joined Relations

- Join operations take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the from clause

Joined Relations

Cartesian with where clause

Select *name, course_id*
from *instructor, teaches*
where *instructor.ID = teaches.ID;*

Joined Relations

Natural Join

Select name, course_id
*from instructor **natural join** teaches;*

Joined Relations

- There is also ***join*** with ***using*** clause

  Select name, course_id

  *from instructor **join** teaches **using** (ID);*
- You must specify list of attributes to join upon
- Both relations must have the same name
- Similar to natural join except:
  - Not all attributes that are the same are joined upon

Joined Relations

- There is also ***join*** with ***on*** condition
  - *Select name, course_id*
  - *from instructor* ***join*** *teaches* ***on*** *(instructor.ID = teaches.ID);*
- Arbitrary join condition
- Similar to using ***where*** clause to specify join condition
  - The ***on*** condition behaves differently for outer joins

Join Example

```
select * from course, prereq where
course.course_id = prereq.course_id;
```

```
+-----------+-------------+------------+---------+-----------+-----------+
| course_id | title       | dept_name  | credits | course_id | prereq_id |
+-----------+-------------+------------+---------+-----------+-----------+
| BIO-301   | Genetics    | Biology    |       4 | BIO-301   | BIO-101   |
| CS-190    | Game Design | Comp. Sci. |       4 | CS-190    | CS-101    |
+-----------+-------------+------------+---------+-----------+-----------+
2 rows in set (0.00 sec)
```

Join Example

select * from course natural join prereq;

```
+-----------+-------------+------------+---------+-----------+
| course_id | title       | dept_name  | credits | prereq_id |
+-----------+-------------+------------+---------+-----------+
| BIO-301   | Genetics    | Biology    |       4 | BIO-101   |
| CS-190    | Game Design | Comp. Sci. |       4 | CS-101    |
+-----------+-------------+------------+---------+-----------+
2 rows in set (0.00 sec)
```

Join Example

```
select * from course join prereq
using(course_id);
```

```
+-----------+-------------+------------+---------+-----------+
| course_id | title       | dept_name  | credits | prereq_id |
+-----------+-------------+------------+---------+-----------+
| BIO-301   | Genetics    | Biology    |       4 | BIO-101   |
| CS-190    | Game Design | Comp. Sci. |       4 | CS-101    |
+-----------+-------------+------------+---------+-----------+
2 rows in set (0.00 sec)
```

Join Example

select * from course join prereq on course.course_id = prereq.course_id;

```
+-----------+-------------+------------+---------+-----------+-----------+
| course_id | title       | dept_name  | credits | course_id | prereq_id |
+-----------+-------------+------------+---------+-----------+-----------+
| BIO-301   | Genetics    | Biology    |       4 | BIO-301   | BIO-101   |
| CS-190    | Game Design | Comp. Sci. |       4 | CS-190    | CS-101    |
+-----------+-------------+------------+---------+-----------+-----------+
2 rows in set (0.01 sec)
```

Outer Joins

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses null values.
- ***inner join*** – join operations that do not preserve non-matched tuples

Left Outer Join

select * from course ***natural left outer join*** prereq;

```
+-----------+-------------+------------+---------+-----------+
| course_id | title       | dept_name  | credits | prereq_id |
+-----------+-------------+------------+---------+-----------+
| BIO-301   | Genetics    | Biology    |       4 | BIO-101   |
| CS-190    | Game Design | Comp. Sci. |       4 | CS-101    |
| CS-315    | Robotics    | Comp. Sci. |       3 | NULL      |
+-----------+-------------+------------+---------+-----------+
3 rows in set (0.00 sec)
```

Right Outer Join

select * from course ***natural right outer join*** prereq;

```
+-----------+-----------+-------------+------------+---------+
| course_id | prereq_id | title       | dept_name  | credits |
+-----------+-----------+-------------+------------+---------+
| BIO-301   | BIO-101   | Genetics    | Biology    |       4 |
| CS-190    | CS-101    | Game Design | Comp. Sci. |       4 |
| CS-347    | CS-101    | NULL        | NULL       |    NULL |
+-----------+-----------+-------------+------------+---------+
3 rows in set (0.00 sec)
```

Full Outer Join

select * from course ***natural full outer join*** prereq;

```
+-----------+-------------+-------------+------------+-----------+
| course_id | title       | dept_name   | credits    | prereq_id |
+-----------+-------------+-------------+------------+-----------+
| BIO-301   | Genetics    | Biology     | 4          | BIO-101   |
| CS-190    | Game Design | Comp. Sci.  | 4          | CS-101    |
| CS-315    | Robotics    | Comp. Sci.  | 3          | NULL      |
| BIO-301   | BIO-101     | Genetics    | Biology    | 4         |
| CS-190    | CS-101      | Game Design | Comp. Sci. | 4         |
| CS-347    | CS-101      | NULL        | NULL       | NULL      |
+-----------+-------------+-------------+------------+-----------+
6 rows in set (0.00 sec)
```

Midterm Review

A large organization has several parking lots which are used by staff.
Each parking lot has a unique name, location, capacity, and number of floors.
Each parking lot has parking spaces which are uniquely identified within that lot using a space number.
Each space is located on a particular floor within the lot.
Members of staff are assigned the use of a parking space.
Each member of staff has a unique id number, name, telephone, and vehicle license plate number..

Midterm Review

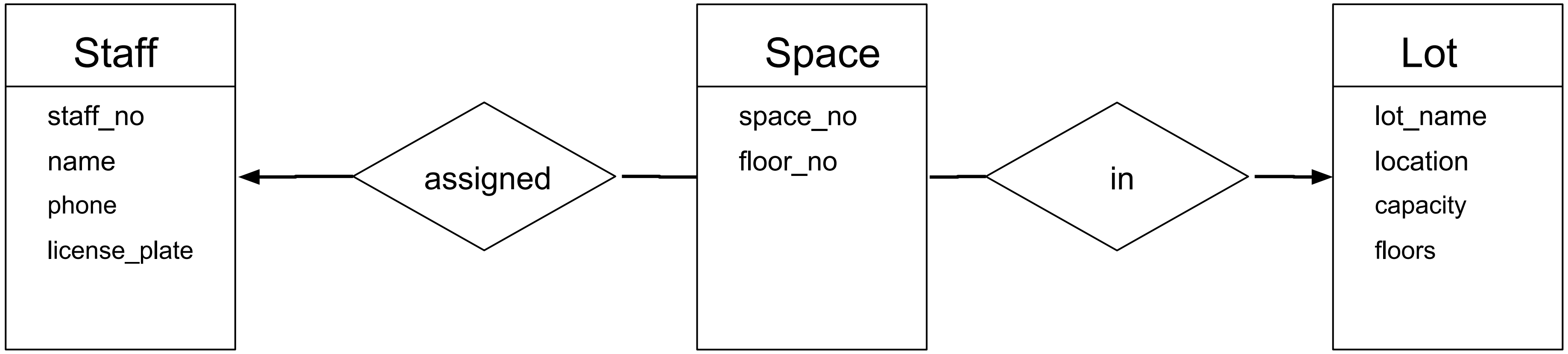
How should we begin?

Midterm Review

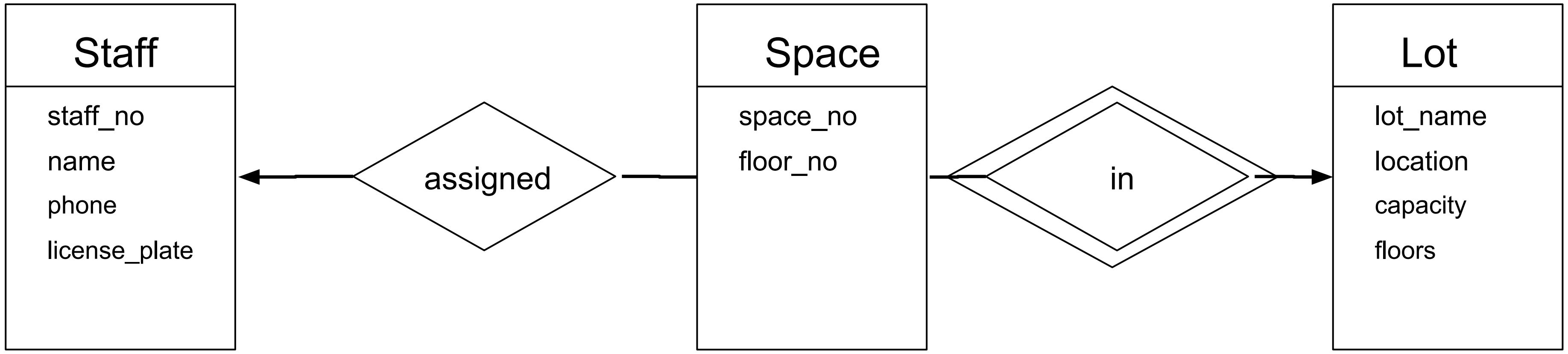
How should we begin?
Create an ER Diagram

What are the entity sets?
What are the relation sets?
Attributes?

Midterm Review



Midterm Review



Midterm Review

Reduce to a relational model.
Underline primary keys and use a + to indicate foreign keys.

Midterm Review

Staff (*staff_no*, first_name, last_name, phone, license_plate)
Assignment(*staff_no(+), lot_name(+), space_number(+)*)
Space (*lot_name(+) , space_number*, floor_number)
Lot (*lot_name*, location, capacity, floors)

Midterm Review

List the set of functional dependencies for each relation in your model that originated from the problem statement above. For each functional dependency, use the form $\alpha \rightarrow \beta$ where α and β are one or more attributes from your relation schema.

Indicate the relation and the set of functional dependencies using this pattern:
Relation name: F= {a→b, c→d }

Midterm Review

Staff:

F = { staff_no→ first_name, last_name, phone, license_plate}

Assignment:

F = { staff_no, lot_name →space_number}

Space:

F = { lot_name , space_number→ floor_number}

Lot:

F = { lot_name → location, capacity, floors }

Midterm Review

Do we pass the test for BCNF?

Why or why not?

Midterm Review

Do we pass the test for BCNF?

Yes, because they are all non-trivial and all superkeys.

Midterm Review

Based on the functional dependencies from above, use Armstrong’s Axioms and their derivatives, to derive three new functional dependencies.

Midterm Review

DECOMPOSITION

lot_name → location, capacity, floors

lot_name → location, capacity

lot_name → floors

REFLEXIVITY

staff_no→ staff_no

AUGMENTATION

staff_no→ license_plate

staff_no, phone → license_plate, phone