Relational Model

- relational model is based on the concept relation or table
- a (mathematical) relation on $D_1, D_2, .. D_n$ is a subset of the **Cartesian product**

Terminology

word	definition
relation	table
attribute	column
tuple	row
arity of a relation	number of **attributes**
cardinality of a relation	number of **tuples**

- a database table is a relation
- **schema** - the **structure** of a relation
  - change rarely
- **instance** - the **content** of a relation
  - change constantly

Schema

Teams(Name, HomeField, Coach)

Instance

Name	Home Field	Coach
Rangers	Runnymede CI	Tarvo Sinervo
Ducks	Humber Public	Tracy Zheng
Choppers	High Park	Ammar Jalali
Crullers	WTCS	Anna Liu

- databases usually store the current version of the data
  - record the history → temporal database

Terminology

Name	Home Field	Coach
Rangers	Runnymede CI	Tarvo Sinervo
Ducks	Humber Public	Tracy Zheng
Choppers	High Park	Ammar Jalali
Crullers	WTCS	Anna Liu

- relations are set
  - can be no duplicate tuples
  - order of the tuples doesn't matter

Key

- characteristics - if for a set of attributes $a_1, a_2, \ldots, a_n$
  - **uniqueness** - their combined values must be unique
  - **minimal** - no subset of $a_1, a_2, \ldots, a_n$ has this property
- use **underline** to declare a key constrains
- **superkey** - any superset of a key
  - not necessarily a proper superset
- means **in principle** there cannot be any duplicates in a particular instance of the relation
- often invent an attribute to ensure all tuples will be unique → SIN, ID
- defines a kind of **integrity constraint**

foreign key

- references - relation refer to each other
- foreign key - **referring attribute**
  - refers to an attribute that is a key in another table
- use **R[A]** to declare
  - R is a relation and A is a list of attributes in R
  - R[A] is the set of all tuples from R but with only the attributes in list A

Artists

aID	aName	nat
1	Nicholson	American
2	Ford	American
3	Stone	British
4	Fisher	American

Artists[aName, nat]

aName	nat
Nicholson	American
Ford	American
Stone	British
Fisher	American

Artists[nat]

nat
American
British

- declare foreign key constraints → $R_1[X] \subseteq R_2[Y]$

    - X and Y may be lists of attributes
    - Y must be a key in R_2
- **referential integrity constraint**
    - foreign key constraints is a kind (subset) of
        - iff Y is a key for relation R_2

Relational Algebra

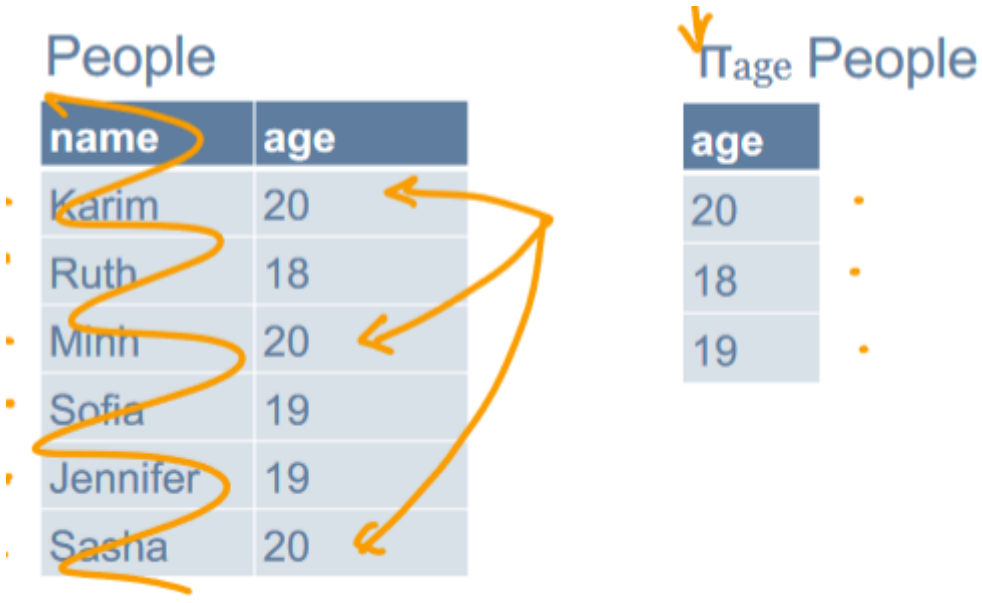
Choose

Select $\sigma_c(R)$ - choose <u>rows</u>

- R - table
- condition c - boolean expression
- return the same schema but with **only the tuples** that satisfy the condition

Project $\pi_L(R)$ - choose <u>columns</u>

- R - table
- L - subset of the attributes of R
- return **all the tuples** from R but with **only the attributes** in L, and **in that order**
- "introduce" duplicates → **only one** copy of each is kept



Combination

Cartesian Product $R \times S$

- return a relation with every combination of a tuple from R_1 concatenated to a tuple from R_2

profiles:

ID	name
oprah	Oprah Winfrey
ginab	Gina Bianchini

2

follows:

follower	followee
oprah	ev123
edyson	ginab
ginab	ev123

3

profiles X follows: 2X3 =6

ID	name	follower	followee
oprah	Oprah Winfrey	oprah	ev123
oprah	Oprah Winfrey	edyson	ginab
oprah	Oprah Winfrey	ginab	ev123
ginab	Gina Bianchini	oprah	ev123
ginab	Gina Bianchini	edyson	ginab
ginab	Gina Bianchini	ginab	ev123

Natural Join $R \bowtie S$

- how the result be defined
  - taking the Cartesian product
  - selecting to ensure **equality** on attributes that are in **both relations**
  - projecting to **remove duplicate** attributes
- ??? attribute in common
  - no → same as Cartesian product
  - one → the steps mentioned above
  - more than one → all the attributes should be matched

Attribute ID in common

profiles:

ID	name
oprah	Oprah Winfrey
ginab	Gina Bianchini

follows:

ID	followee
oprah	ev123
edyson	ginab
ginab	ev123

ID	name	ID	followee
oprah	Oprah Winfrey	oprah	ev123
oprah	Oprah Winfrey	edyson	ginab
oprah	Oprah Winfrey	ginab	ev123
ginab	Gina Bianchini	oprah	ev123
ginab	Gina Bianchini	edyson	ginab
ginab	Gina Bianchini	ginab	ev123

22

- properties
  - commutative - $R \bowtie S = S \bowtie R$ (attribute order may vary)
  - associative - $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T)$
- if no tuples match → <u>empty set</u>

Theta Join $R \bowtie_{condition} S$

- use σ to check conditions <u>after a Cartesian product</u>
- same as Cartesian product followed by select
  - $R \bowtie_{condition} S = \sigma_{condition}(R \times S)$

Other operation

Assignment $R(A_1, \ldots, A_n) := Expression$

- name all the attributes of the new relation
- R must be a **temporary variable**, not one of the relations in the schema

Rename $\rho_{R_1}(R_2)$

- alternate notation $\rho_{R_1(A_1,\ldots,A_n)}(R_2)$ → rename all the attributes as well as the relation
- following are equivalent
  - $R_1(A_1, \ldots, A_2) := R_2$
  - $R_1 := \rho_{R_1(A_1,\ldots,A_n)}(R_2)$

Summary

- Precedence - from highest to lowest

σ, π, ρ

$\times$, $\bowtie$

$\cap$

$\cup$, $-$

Operation	Name	Symbol
choose rows	select	σ
choose columns	project	π
combine tables	Cartesian product	×
	natural join	⋈
	theta join	$\bowtie_{condition}$
rename relation [and attributes]	rename	ρ
assignment	assignment	:=

Technique

find max/min

- find the tuple is **not** the max/min

$$\text{– sID did not have the highest grade in csc343 in term 20099.}$$
$$NotTop(sID) := \Pi_{T1.sID}\sigma_{T1.grade<T2.grade}[(\rho_{T1}Takers) \times (\rho_{T2}Takers)]$$

- then **minus** the tuples not satisfies

$$Answer(sID) := (\Pi_{sID}Takers) - NotTop$$

find occurrence more than one

- self Cartesian product, and select the tuples with different value for the specific attribute

$$\text{– NID of all multi-object donations.}$$
$$MultiObjectNID(NID) := \Pi_{C1.NID}\sigma_{C1.NID=C2.NID \wedge C1.CN \neq C2.CN}[(\rho_{C1}Contains) \times (\rho_{C2}Contains)]$$

→ find **only one** occurrence

- minus multiple occurrence and **no occurrence**

– SID who catalogues the objects.
$CatStaff(who) := \Pi_{who} Object$

– SID catalogued objects with at least one tag.
$AtLeastOne(CN, who, phrase) := \Pi_{CN,who,phrase} Object \bowtie Tag$

– SID does not use more than one tag for all objects.
$OneTag(who) := CatStaff -$
$\Pi_{O1.who}[(\Pi_{O1.CN}\sigma_{O1.CN=O2.CN \wedge O1.phrase \neq O2.phrase}[(\rho_{O1} AtLeastOne) \times (\rho_{O2} AtLeastOne)]$

– Objects catalogued by the staff who do not use more than one tag for all objects.
$OneTagObjects(CN) := \Pi_{CN}(OneTag \bowtie Object)$

– Objects catalogued by the staff who does not use any tag for at least one object.
$ZeroTag(CN) := OneTagObjects - \Pi_{CN}(OneTagObjects \bowtie Tag)$

– Staff who use exact one tag for all objects.
$Answer(SID) := OneTag - \Pi_{who}(ZeroTag \bowtie Object)$

find match everything

- find the **expected** relation with Cartesian product

  – Expected SID experting in every Chenhall category.
  $Expected(SID, sec, prim, cate) := NotSup \times FullCategory$

- find the **actual** relation

  – Actual SID for every secondary term.
  $Actual(SID, sec, prim, cate) := \Pi_{who,type,ptim,cate}\sigma_{type=sec}[(\Pi_{who,type} Object) \times FullCategory]$

- expected minus the tuples that are not in actual

  – SID expert in one or more Chenhall category.
  $Expert(SID) := \Pi_{SID}[\Pi_{SID,cate} Expected - \Pi_{SID,cate}(Expected - Actual)]$

SQL

Basic Operation

SELECT-FROM-WHERE

- SELECT → projection π
- WHERE → select σ
- FROM → table

```
SELECT <attribute>
FROM <table_name>
WHERE <condition>;
```

- if we put **more than one** table in a comma-separated list in the FROM clause
  - we will get the **Cartesian product** of these tables

```
SELECT <attribute>
FROM <table_1>, <table_2>, <table_3>
WHERE <condition>;
```

- we can use basically logical operators in WHERE clause
  - != and <> mean "not equals"
- wildcard * in SELECT → select **all**
- can use an expression in a SELECT clause
  - algebra
  - string concatenation || → default column name "?column?"
  - constant

```
SELECT sid, grade + 10 AS adjusted
FROM Took;

SELECT dept || cnum
FROM Course
```

```
WHERE cnum < 200;

SELECT name, 'satisfies' as breadthRequirement
FROM Course
WHERE breadth;
       name        | breadthrequirement
-------------------+--------------------
 Intro Archaeology | satisfies
 Narrative         | satisfies
 Rhetoric          | satisfies
 The Graphic Novel | satisfies
 Mediaeval Society | satisfies
 Black Freedom     | satisfies
```

Rename

- temporarily renaming
  - add the new name between the table name and the comma

```
SELECT <attribute>
FROM <table_1> <rename_1>, <table_2> <rename_1>
WHERE <condition>;

SELECT e.name, d.name
FROM Employee e, Department d
WHERE d.name = 'marketing' and e.name = 'Horton';
```

- naming columns → add **AS**-expression to choose a column name

```
SELECT <attribute> AS <new_name>
FROM <table_name>
WHERE <condition>;
```

Sorting

- use **ORDER BY**- expression, default ascending order

```
SELECT <attribute>
FROM <table_name>
WHERE <condition>
ORDER BY <attribute>;
```

- override the order by add **DESC**

```
SELECT <attribute>
FROM <table_name>
WHERE <condition>
ORDER BY <attribute> DESC;
```

- order according to multiple columns
  - first sort by the **first** attribute, if same, then by the **second**

```
SELECT <attribute> AS <new_name>
FROM <table_name>
WHERE <condition>
ORDER BY <attribute_1>,  <attribute_2>
```

Pattern Matching

- _ - matches any **single** character
  - `UT_` → 'UTM' ✓, 'UTSG' ×
- `%` - matches any **sequence of zero or more** character
  - `%to%` → any string contains the substring 'to'

- LIKE - offers an extremely limited form of pattern matching but has the advantage of being fast.

```
SELECT *
FROM Course
WHERE name LIKE '%to%';
 cnum |           name             | dept | breadth
------+----------------------------+------+---------
  343 | Intro to Databases         | CSC  | f
  148 | Intro to Comp Sci          | CSC  | f
  320 | Intro to Visual Computing | CSC  | f
  263 | Compar Vert Anatomy        | EEB  | f
  205 | Rhetoric                   | ENG  | t
(5 rows)

SELECT *
FROM Student
WHERE campus LIKE 'UT_';
 sid | firstname |  surname  | campus |  email  | cgpa
-----+-----------+-----------+--------+---------+------
 157 | Leilani   | Lakemeyer | UTM    | lani@cs | 3.42
```

- can use ~ operator

```
-- Find students whose surname begins with M or F or L, followed
-- by 'a', and then zero or more additional characters.
SELECT *
FROM Student
WHERE surname ~ '(M|F|L)a*';
  sid  | firstname |  surname   | campus |  email   | cgpa
-------+-----------+------------+--------+----------+------
 99132 | Avery     | Marchmount | StG    | avery@cs | 3.13
 98000 | William   | Fairgrieve | StG    | will@cs  | 4.00
   157 | Leilani   | Lakemeyer  | UTM    | lani@cs  | 3.42
```

Aggregation and Grouping

Aggregation

- functions such as `sum`, `avg`, `min`, `max` , `count`
  - can be used in a SELECT clause to apply to a column
  - default column name is the function name
- if operate on the result two aggregation, the default column name is `?column?`

```
SELECT max(grade) - min(grade)
FROM Took;
 ?column?
----------
      100
```

- `count(*)` - count the number of rows in the table
- use `DISTINCT` to introduce the duplicate

```
SELECT count(dept)
FROM Offering;
 count
-------
    36

SELECT count(DISTINCT dept)
FROM Offering;
 count
-------
     6
```

- include more than one aggregation in a query use comma to separate
  - should dealing with like quantities (#result are same)

Grouping

- GROUP BY - make the tuples that have the same value for that attribute will be treated as a group
  - **one row will be generated for each group**
  - all attribute **must** be grouped or in a aggregation

```
SELECT <attribute_1>, aggregation(<attribute_2>)
FROM Took
GROUP BY <attribute_1>;

SELECT sid, grade
FROM Took
GROUP BY sid;
ERROR:  column "Took.grade" must appear in the GROUP BY clause or be used
in an aggregate function
LINE 1: SELECT sid, grade
                    ^

SELECT sid, avg(grade)
FROM Took
GROUP BY sid;
  sid  |         avg
-------+---------------------
 11111 | 29.6000000000000000
 98000 | 83.2000000000000000
 99132 | 76.2857142857142857
 99999 | 84.5833333333333333
   157 | 75.9333333333333333
```

- can order by attributes that are not in the output
  - but **must in aggregation** → cannot guarantee each group has this attribute
- group multiple attribute → group by the pair

Having

- filtering Groups with `HAVING` expression → similar to WHERE

```
SELECT <attributes>
FROM <table_name>
GROUP BY <attributes>
HAVING <conditions>
```

- can **only** refer an aggregated or grouped attribute

```
SELECT oid, avg(grade), count(*)
FROM Took
GROUP BY oid
HAVING grade < 50;
ERROR:  column "took.grade" must appear in the GROUP BY clause or be used
in an aggregate function
LINE 4: HAVING grade < 50;
               ^
```

Order

FROM > WHERE > GROUP BY > HAVING > SELECT > ORDER BY

VIEW

- ask SQL to remember a definition

```
CREATE VIEW <view_name> AS
SELECT
FROM
...
```

- always update → **recomputed every time** they are used

Duplication

- table in SQL **can** have duplicate rows
  - when the primary key is unique
- SELECT-FROM-WHERE under bag semantics → produces **duplicate** rows
- use DISTINCT to override the default behaviour
  - only remove the **entirely repeated** tow
  - the partial duplications **will not** be removed
- **only** ask for distinct **rows**, **not** by **column**

```
SELECT oid, grade
FROM Took
WHERE grade > 95
ORDER BY oid;
 oid | grade
-----+-------
   1 |    99
  11 |    99          <-- (a) These rows are duplicates
  11 |    99          <-- (same oid and same grade)
  13 |    99
  13 |    98          <-- (b) These rows have the same grade,
  14 |    98          <-- but different oids.
  16 |   100          <-- (c) These rows have the same oid,
  16 |    98          <-- but different grades
  22 |    96

SELECT DISTINCT oid, DISTINCT grade
FROM Took
WHERE grade > 95
ORDER BY oid;
ERROR:  syntax error at or near "DISTINCT"
LINE 1: SELECT DISTINCT oid, DISTINCT grade FROM Took WHERE grade > ...
                             ^
```

set operation

∪ UNION , ∩ INTERSECT, \ EXCEPT

- under set semantics → no duplication
  - change default use ALL → contains duplication

```
(SELECT sid FROM Took WHERE grade > 95)
UNION ALL
(SELECT sid FROM Took WHERE grade < 50);
```

- an operand to a set operator must be surrounded by **round brackets**
- the two operand to a set operator must have **compatible schemas**
- an operand to a set operator must be a **complete query**
  - `(Elephant) INTERSECT (Flower)` → WRONG
  - `(SELECT * FROM Elephant) INTERSECT (SELECT * FROM Flower)`
- EXCEPT v.s EXCEPT ALL
  - EXCEPT → minus **all**
    - still give you a set
  - EXCEPT ALL → just minus one times

```
SELECT *
FROM P;
 a |  b
---+-----
 1 | 151
 2 | 123
 3 | 432
 1 | 333
 1 | 345
 4 | 912
 5 | 123
```

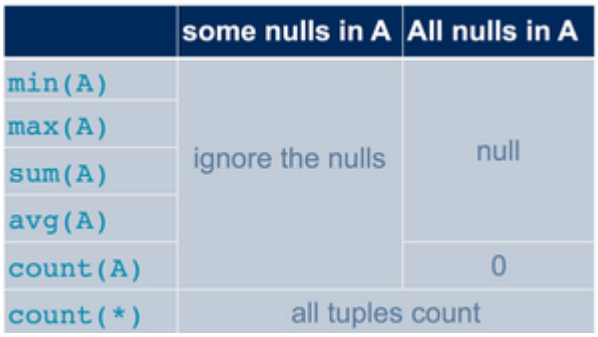
```
(7 rows)

SELECT *
FROM Q;
 a | c
---+----
 1 | 44
 3 | 88
 3 | 12
 9 | 12
(4 rows)

(SELECT a FROM P)
EXCEPT ALL
(SELECT a FROM Q);
 a
---
 1               <-- There are two 1s in this result
 1
 2
 4
 5
(5 rows)
```

But with EXCEPT, a single occurence of a value for a in Q wipes out *all* occurences of it from P.
So if we use EXCEPT in our query from before, every 1 value in P is removed as a result of a single 1 value in Q.

```SQL
(SELECT a FROM P)
EXCEPT
(SELECT a FROM Q);
 a
---
 2
 4               <-- There are no 1s anywhere in this result
 5
(3 rows)
```

Null Value

- `NOT NULL` constraint - prevent the column to be null
- `IS` caparison - IS NULL, IS NOT NULL
- unknown - if one or both operands is `NULL`
- on **WHERE** & **NATURAL JOIN**
  - will **not** include a row
- on **aggregation**
  - `NULL` **never** contributes to a `sum`, `avg`, or `count`



	some nulls in A	All nulls in A
min(A)	ignore the nulls	null
max(A)		
sum(A)		
avg(A)		
count(A)		0
count(*)	all tuples count	

```
name  | age | grade
------+-----+-------
 diane |     |     8
 will  |     |     8
 cate  |     |     1
 tom   |     |
 micah |     |     1
 grace |     |     2

 count(age) -> 0
 count(*) -> 6
```

- on **boolean** conditions

A	B	A and B	A or B
T	T	T	T
TF or FT		F	T
F	F	F	F
TU or UT		U	T
FU or UF		F	U
U	U	U	U

A	not A
T	F
F	T
U	U

- **tautology isn't true for NULL**

```
SELECT *
FROM Ages;
  name   | age
---------+-----
 Amna    |  21
 Zach    |  25
 Miriam  |
 Ben     |   0
(4 rows)

SELECT *
FROM Ages
WHERE age >= 10 OR AGE < 10;
 name | age
------+-----
 Amna |  21
 Zach |  25
 Ben  |   0
(3 rows)
```

Join

NATURAL JOIN

- dangling tuples - not represent in the result
  - e.g. $< 4, 5 >, < 6, 7 >$

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL JOIN S

A	B	C
1	2	3

Outer Join

- preserve dangling tuples by padding them with **null** values where needed
- inner join - the join don't pad with nulls
- **left** outer join - preserve dangling tuples from the table on the **LHS** only

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL LEFT JOIN S

A	B	C
1	2	3
4	5	NULL

- **right** outer join - preserve dangling tuples from the table on the **RHS** only

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL RIGHT JOIN S

A	B	C
1	2	3
NULL	6	7

- **full** outer join - does both

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL FULL JOIN S

A	B	C
1	2	3
4	5	NULL
NULL	6	7

- SQL syntax

Cartesian product

```
A CROSS JOIN B                    same as A,B
```

Theta-join

```
A JOIN B ON C
✓A {LEFT|RIGHT|FULL} JOIN B ON C
```

Natural join

```
A NATURAL JOIN B
✓A NATURAL {LEFT|RIGHT|FULL} JOIN B
```

✓ indicates that tuples are padded when needed.

- if wish to use an outer join, **must** use `JOIN ON`

- `JOIN ON` is better than `NATURAL JOIN` → more clear

- JOIN USING - similar to natural join but will **forced to match**
  - no longer need to rename

Subqueries

in FROM

- take the place of a **table name**
- must be enclosed in **round brackets**
- must **name** the result of the subquery

```
SELECT sID, dept||cNum as course
FROM Took,
(SELECT *                          -- subqueriees
FROM Offering
WHERE instructor = 'Horton'
) Hoffering
WHERE Took.oID = Hoffering.oID;
```

in WHERE

- if produces **exactly one row**, we can compare to it in a WHERE clause

```
SELECT sID, surname, cgpa
FROM Student
WHERE cgpa >
(SELECT cgpa
FROM Student
WHERE sID = 99999);
```

- subquery return **empty** table → produce an **empty table** finally
- subquery return **more than one** row → ERROR

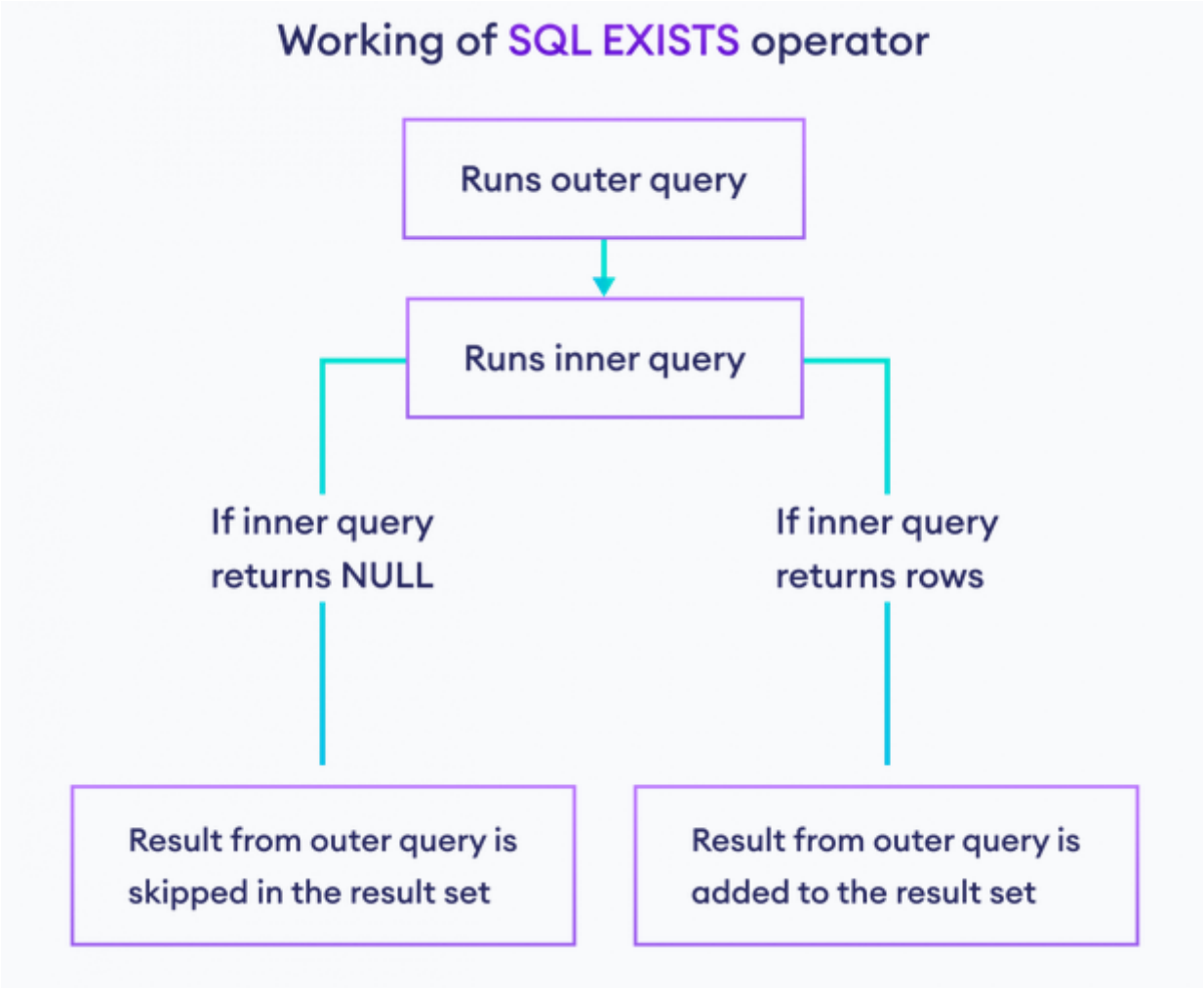
- use **quantifier** - ALL, SOME, ANY

quantifier

- `ALL` - ps. consider the "equal" situation
- `SOME` - hold for **at least one**
  - synonym `ANY` - easily misleading
- `IN` and `NOT IN` - want to know whether or not a particular value occurs in it
  - `IN` equal to `=SOME`
  - `NOT IN` equal to `<> ALL`

```
SELECT sID, dept||cnum AS course, grade
FROM Took JOIN Offering USING (oID)
WHERE oID IN (                              -- same to use = SOME
    SELECT oID
    FROM Offering
    WHERE instructor = 'Atwood'
);

SELECT sID, dept||cnum AS course, grade
FROM Took JOIN Offering USING (oID)
WHERE oID NOT IN (                          -- same to use <> ALL
    SELECT oID
    FROM Offering
    WHERE instructor = 'Atwood'
);
```

- `EXISTS` - is true iff the subquery has **at least one** tuple
  - like a for loop → each row run the subquery
  - if true → will be a row in the result
  - the subquery must be recomputed → **correlated subquery**
  - `NOT EXISTS`



x *«comparison»* ALL (*«subquery»*)
∀ y ∈ *«subquery results»* | x *«comparison»* y

x *«comparison»* SOME (*«subquery»*)
∃ y ∈ *«subquery results»* | x *«comparison»* y

x IN (*«subquery»*)
Same as x = SOME (*«subquery»*)

x NOT IN (*«subquery»*)
Same as x <> ALL (*«subquery»*)

just for convenience

EXISTS (*«subquery»*)
∃ y ∈ *«subquery results»*

Database Modification

Insertion

```
CREATE TABLE <table_name>(<attr_1>, <attr_2>,...);

INSERT INTO <table_name> VALUES
(<tuple_1>), (<tuple_2>), ...;

INSERT INTO <table_name>
(SELECT ...
FROM ...
WHERE ...);
```

- with **default** value

```
CREATE TABLE Invite (
    name TEXT,
    campus TEXT DEFAULT 'STG',
    age INT
);

INSERT INTO Inivte(name)
(SELECT firstname,
FROM Student
WHERE cgpa > 3.4);

SELECT *
FROM Invite;
  name   | campus | age
---------+--------+-----
 William | StG    |
 Leilani | StG    |
```

Deletion

```
DELETE FROM <table_name>
WHERE <condition>;

DELETE FROM <table_name>;   -- delete the table
```

Update

- UPDATE table SET list of attribute assignments WHERE condition on tuples

```
UPDATE <table_name>
SET <attribut> = <value>
WHERE ...;
```

Embedded SQL

Stored Procedures

- have parameters and a return value
- use local variables, ifs, loops, etc.
- execute SQL queries
- benefit - **efficient**

- A boolean function `QuietYear(y INT, s CHAR(15))` that returns true iff
  - movie studio s produced no movies in year y, or
  - produced at most 10 comedies.

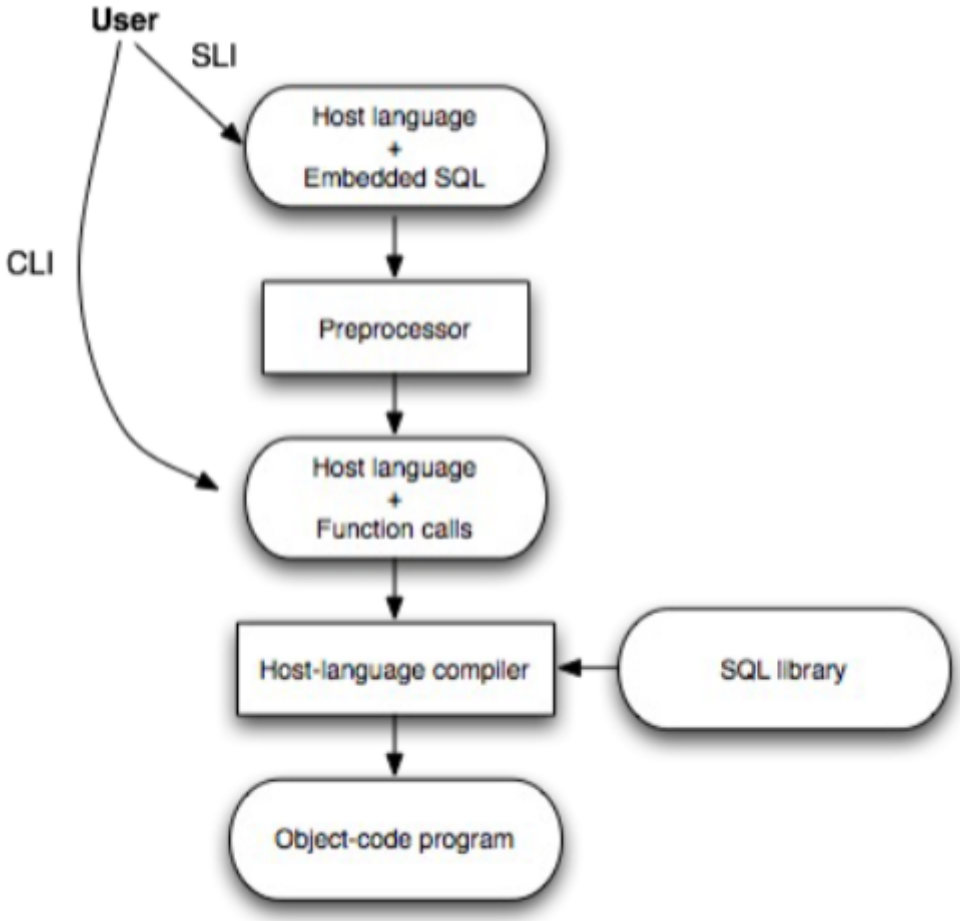
Reference: textbook figure 9.1.3 (example slightly modified here)

```
CREATE FUNCTION QuietYear(y INT, s CHAR(15)) RETURNS BOOLEAN
IF NOT EXISTS
   (SELECT *
    FROM Movies
    WHERE year = y AND studioName = s)
THEN RETURN TRUE;
ELSIF 10 <=
   (SELECT COUNT(*)
    FROM Movies
    WHERE year = y AND studioName = s AND
          genre = 'comedy')
THEN RETURN TRUE;
ELSE RETURN FALSE;
END IF;
```

```
SELECT StudioName
FROM Studios
WHERE QuietYear(2010, StudioName);
```

Statement-level Interface (SLI)

- embed SQL statements into code in conventional language
- use a preprocessor to replace the SQL with calls written in the host language to functions defined in an SQL library



Call-level Interface (CLI)

- write those calls yourself
- eliminates need to preprocess

static query

- import library

```
import psycopg2 as pg
```

- create an object that holds a connection to your database

```
conn = pg.connect(
    dbname="", user="", password="",
    options="-c search_path=university,pu
)
```

- open cursor object that can
  - **hold** the results of a query
  - allow us to **iterate** over them

```
cur = conn.cursor()
```

- **execute** a SQL query and store the results in cursor (with try)

```
try:
    cur.execute("<SQL sentence>")

    # iterate over those results
    for record in cur:
        ...
```

- **handle the error** by rolling back

```
except pg.Error as ex:
    conn.rollback
```

- **close** cursor and connection (very **end** of the program)

```
finally:
    cur.close()
    conn.close
```

dynamic query

- if query depends on something → e.g. result of a computation
- same as static, but for execution
  - use **f-string** to generate SQL
  - if make some change, should **commit**

```
try:
    print("...")
    <var_1> = input("...")
    <var_2> = input("...")
    <var_3> = input("...")

    cursor.execute(f"...{<var_1>}...{<var_2>}...{<var_3>}...")

    #commit the change to the database -> insert, update,...
    conn.commit()
```

Safe

- injection attack -
- sending a **separate parameter** to the execute method is safe
- others → computing the complete string in Python and send the complete string to execute → risky

```
# Sending a separate parameter to the execute method is safe:
cursor.execute("INSERT INTO COURSE VALUES (%s, %s, %s);", [cnum, name,
dept])
```

Data Definition Language

Types

- must define the type of each column when creating a table

built-in types

type	example	description
CHAR(n)	's'	**fixed-length** string of n characters, padded with blanks if necessary
VARCHAR	'asda'	**variable-length** string of **up to** n characters
TEXT	'dfafa'	**variable-length, unlimited**
INT	37	=INTEGER
FLOAT	1.354	=REAL
BOOLEAN	TRUE	true or false
DATE	'2011-09-22'	
TIME	'15:00:02'	
TIMESTAMP	'Jan-12-2011 10:25'	date plus time

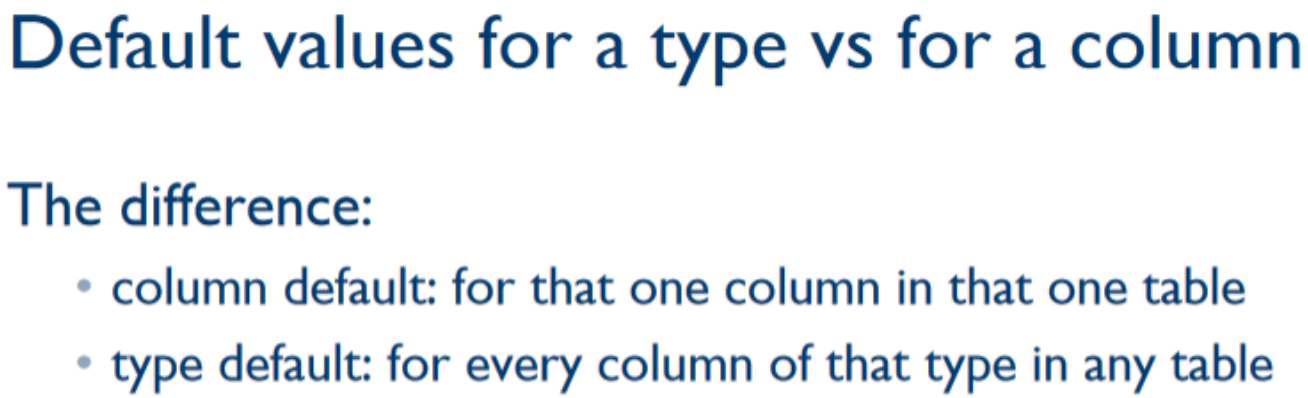
user-defined types

- defined in terms of a built-in type

```
create domain Grade as int
    default null
    check (value>=0 and value <=100);
create domain Campus as varchar(4)
    default 'StG'
    check (value in ('StG','UTM','UTSC'));
```

semantics

- type constraints - checked **every time** a value is assigned to an column of that type
- default value - used when **no value** has been specified



Key

`PRIMARY KEY`

- rules
  - their values will be **unique** across rows
  - their values will **never be null**
  - every table **must have 0 or 1** primary key → cannot declare more than one
- declaring
  - at the end of the table definition
  - for a single-column key, can declare it with the column

```
create table Person (
    ID integer,
    name varchar(25),
    primary key (ID));

create table Person(
    ID integer primary key,
    name varchar(25));

create table Undergrad (
    firstname varchar(25),
    lastname varchar(25),
    campus varchar(3),
    primary key (lastname, campus));
```

`UNIQUE`

- rule
  - their values will be **unique** across rows
  - their values **can** be null
  - can declare **more than one set** of attributes to be `UNIQUE`
- declaration is same as primary key but use the keyword `UNIQUE`
- **no two nulls** are considered **equal**

`FOREIGN KEY`

- `foreign key (<attribute>) reference <table_name>`
- means that column in this table is a foreign key that references the primary key of another table
- rule - must declared either **primary key** or **unique** in the "home" table

Check

- can define a constraint
  - on a **column** → as part of the column's definition

- on the **row** → at the end of the table definition
- **across table** → with assertions or triggers

column-based

- defined with a single column and constrain its value

```
create table <table_name> (
    <att_name> <att_type> check (<condition>)
);
```

- condition can be anything that could go in a WHERE clause
- can include a subquery
- when checked
  - only when a row is **inserted** into that table
  - or its value for that **column** is **updated**
- if change somewhere else violates the constraint→ not notice
- only an issue if the constraint involves a subquery

not null

- declare that a column of a table is NOT NULL

row-based

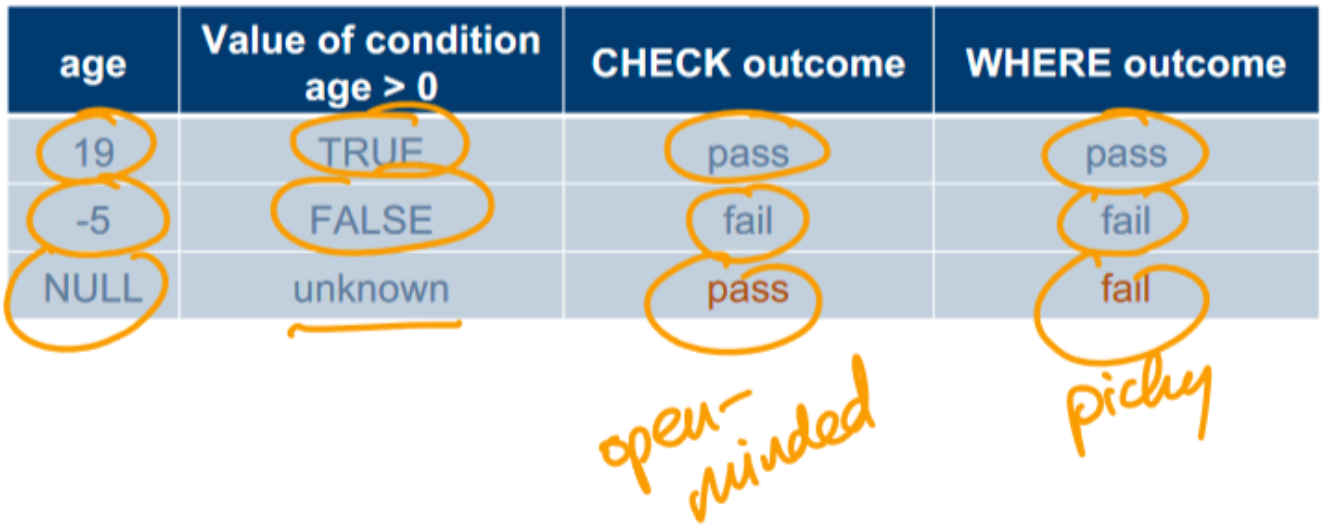
- defined as a separate element of the table schema
  - refer to **any** column of the table

```
create table <table_name>(
    ...
    check <condition>
);
```

- condition's rule same as column based
- when checked
  - only when a row is **inserted** into that table or **update**

null's affect

- check constraint only fails if it evaluates to **false**
- it is **not** picky like a WHERE condition



name

add `constraint <name>` before the `check (<condition>)`

cross-table - assertions

- schema elements at the top level
- can express cross-table constraints
- `create assertion (<name>) check (<predicate>)`
- costly → check every time, **not support by PostgreSQL**

cross-table - triggers

- powerful, control the cost by having control over when they are applied
- basic idea
  - specify a type of database event

- `after ...` or `before ...`

- specify the response

The response function

```
create function RecordWinner() returns trigger
as
$$
BEGIN
    IF NEW.grade >= 85 THEN
        INSERT INTO Winners VALUES (NEW.sid);
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql;
```

The trigger that uses it

```
create trigger TookUpdate
before insert on Took
for each row
execute procedure RecordWinner();
```

Reaction Policies

possible policies

- **cascade** - propagate the change to the referring table
- **set null** - set the referring columns to null
- suppose table R refers to table S
  - **can** define fixes that propogate changes **backwards from S to R**
  - **cannot** define fixes that propogate **forward from R to S**

syntax

- add reaction policy where you specify the **foreign key constraint**

```
create table <table_name> (
    ...
    foreign key ... references ... on <react> <action>
    ...
);
```

- react - on **delete/ update** or both
- action
  - **restrict** - don't allow the deletion/update
  - **cascade** - make the same deletion/update in the referring row
  - **set null** - set the corresponding value in the referring row to null

Schemas

- a kind of namespace
- everything defined (tables, types, etc.) goes into one big pot

create

- already have a schema called public
- create your own

```
create schema <schema_name>;
```

- refer to things inside a particular schema
  - if without specifying what schema it is within → go in **public**

```
create table <schema_name>.<table_name> (...);
select * from <schema_name>.<table_name>;
```

search path

- to see the search path → `show search_path;`
- **set** the search path → `set search_path to <path_1> (, <path_2>,...)`
  - order - path_1 **>** path_2 **>** ...
  - use **comma** to separate the paths
- **default** search path - $user, public
  - `$user` is **not** created for you
  - `public` is created

remove

- `cascade` → everything inside it is dropped too

```
drop schemma <schema_name> cascade;
```

- add `if exists` to avoid getting error if the schema does not exist

pattern

```
drop schema if exists <schema_name> cascade;
create schema <schema_name>;
set search_path to <schema_name>;
```

Design and Normalization

Function Dependency Theory

principle

- avoid **redundancy** - redundant data can lead to anomalies
- **update** anomaly → update **only one** tuple, the data is inconsistent
- **deletion** anomaly → may lose some information

definition

- suppose R is a relation, and X and Y are subsets of the attributes of R
- **X → Y** asserts that
  - if two tuples agree on all the attributes in set X, they **must** also agree on all the attributes in set Y
- X → Y holds in R or X functionally determines Y
- dependency → the value of Y depends on the value of X
- function → takes a value for X and gives a **unique** value for Y
- when write a **set** of FDs → **all** of them hold
- S1 is equivalent to S2 → S1 holds in a relation iff S2 does
- as FD is an assertion about every instance of the relation
- K is a **superkey** for R iff K functionally determines **all** of R
- if there is **no** FDs in a relation, then there is **no** redundancy

R(A, B, C, D)

- Superkey:
  $X \rightarrow R$ — Every attribute

B → ACD
every

- Functional dependency:
  $X \rightarrow Y$ — Not necessarily every attribute

A → CD
can be less than every

- An FD can be more subtle.

multi FDs

$A \rightarrow B$ means:

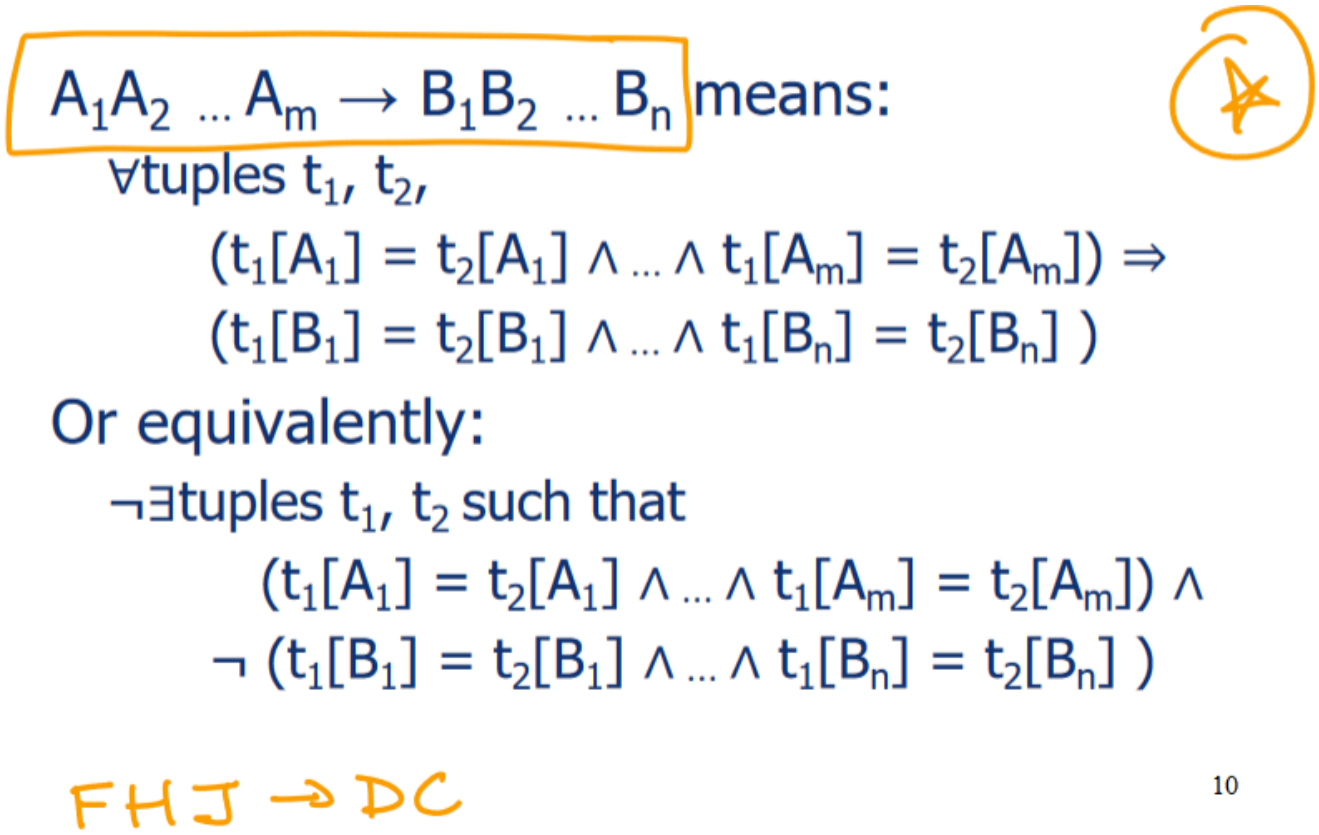
$\forall$tuples t_1, t_2,

$(t_1[A] = t_2[A]) \Rightarrow (t_1[B] = t_2[B])$

Or equivalently:

$\neg\exists$tuples t_1, t_2 such that

$(t_1[A] = t_2[A]) \wedge (t_1[B] \neq t_2[B])$



- we **can** split the **RHS** of an FD and get multiple, equivalent FDs

> $$PQ \rightarrow RST \equiv PQ \rightarrow R, \quad PQ \rightarrow S, \quad PR \rightarrow T.$$

- we **cannot** split the **LHS** of an FD and get multiple, equivalents FDs

inferring

- using **closure test**
  - figure out every thing else that is determined
  - if includes the RHS attributes, then you know that LHS → RHS

Attribute_closure(Y, S):

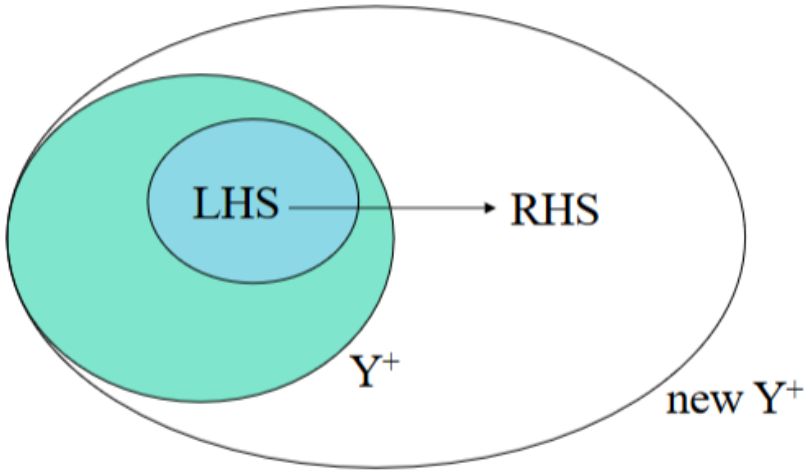
Initialize Y^+ to Y

Repeat until no more changes occur:

If there is an FD LHS $\rightarrow$ RHS in S such that LHS is in Y^+:

Add RHS to Y^+

Return Y^+



If LHS is in Y^+ and LHS $\rightarrow$ RHS holds, we can add RHS to Y^+

- If $A \rightarrow B$ and $B \rightarrow C$ hold,
  must $A \rightarrow C$ hold? yes
- If $A \rightarrow H$, $C \rightarrow F$, and $FG \rightarrow AD$ hold,
  must $FA \rightarrow D$ hold? FAH no.
  must $CG \rightarrow FH$ hold? CGFADH yes.
- If $H \rightarrow GD$, $HD \rightarrow CE$, and $BD \rightarrow A$ hold,
  must $EH \rightarrow C$ hold? EHGDCE yes.
- Aside: we are not generating new FDs, but testing a specific possible one.

FA+ = FAH.

EH+ = EHGDCE.

projecting

S is a set of FDs; L is a set of attributes.
Return the projection of S onto L:
all FDs that follow from S and involve only attributes from L.

Project(S, L):
  Initialize T to {}.
  For each subset X of L:
    Compute X^+ *Close X and see what we get.*
    For every attribute A in X^+:
      If A is in L: *$X \rightarrow A$ is only relevant if A is in L (we know X is).*
        add $X \rightarrow A$ to T.
  Return T.

Suppose we have these FDs: : $S = \{ABE \to CF,\ \ DF \to BD,\ \ C \to DF,\ \ E \to A,\ \ AF \to B\}$

Project the FDs onto: $L = CDEF$

Attributes to take all subsets X of: C D E F	Closure of the subset X^+	Functional dependencies inferred

Solution:

C	D	E	F	closure	FDs
✓				$C^+ = CDFBD$	$C \to DF$
	✓			$D^+ = D$	nothing
		✓		$E^+ = EA$	nothing
			✓	$F^+ = F$	nothing
✓	✓			$CD^+ = CDFB$	nothing, since $CD \to DF$ is weaker than $C \to DF$ which we have already
✓		✓		$CE^+ = CEDFAB$	nothing, since $CE \to DF$ is weaker than $C \to DF$ which we have already
✓			✓	$CF^+ = CFDB$	nothing, since $CF \to D$ is weaker than $C \to DF$ which we have already
	✓	✓		$DE^+ = DEA$	nothing
	✓		✓	$DF^+ = DFB$	nothing
		✓	✓	$EF^+ = EFABCD$	$EF \to CD$
✓	✓	✓		$CDE^+ = CDEF$	nothing, since $CDE \to F$ is weaker than $C \to DF$ which we have already
✓	✓		✓	$CDF^+ = CDFB$	nothing
✓		✓	✓	since EF is a key, supersets of EF can only yield FDs that are weaker than ones we have.	
	✓	✓	✓	since EF is a key, supersets of EF can only yield FDs that are weaker than ones we have.	

Final answer: The projection of S onto L is $C \to DF$, $EF \to CD$.

- no need to add X → A if A is **in** X itself
- no need to compute the closures of **empty set, the set of all attributes**
- if we find that X^+ = **all** attributes, we can **ignore** any **superset** of X

minimal basis

- a set of FDs that is equivalent, but has
  - **no redundant** FDs, and
  - **no** FDs with **unnecessary attributes** on the LHS
- after identify a redundant FD, must **not** use it when computing subsequent closures

Minimal_basis(S):

1. Split the RHS of each FD
2. For each FD $X \rightarrow Y$ where $|X| \geq 2$:

   If you can remove an attribute from X and get an FD that follows from S:

   Do so! (It's a stronger FD.)
3. For each FD f:

   If S – {f} implies f:

   Remove f from S.

. Find a minimal basis for this set of FDs: $S = \{ABF \to G,\ \ BC \to H,\ \ BCH \to EG,\ \ BE \to GH\}$.

Solution:

Step 1: Split the RHSs to get our initial set of FDs, $S1$:

(a) $ABF \to G$
(b) $BC \to H$
(c) $BCH \to E$
(d) $BCH \to G$
(e) $BE \to G$
(f) $BE \to H$.

Step 2: For each FD, try to reduce the LHS:

(a) $A^+ = A, B^+ = B, F^+ = F$. In fact, no singleton LHS yields anything. $AB^+ = AB, AF^+ = AF$, and $BF^+ = BF$, so none of them yields G either. We cannot reduce the LHS of this FD.

(b) Since this FD has only two attributes on the LHS, and no singleton LHS yields anything, we cannot reduce the LHS of this FD.

(c) Since no singleton LHS yields anything, we need only consider LHSs with two or more attributes. We only have three to begin with, so that leaves LHSs with two attributes. $BC^+ = BCHEG$. So we can reduce the LHS of this FD, yielding the new FD: $BC \to E$.

(d) By the same argument, we can reduce this FD to: $BC \to G$.

(e) Since no singleton LHS yields anything, we cannot reduce the LHS of this FD.

(f) Since no singleton LHS yields anything, we cannot reduce the LHS of this FD.

Our new set of FDs, let's call it $S2$, is

(a) $ABF \to G$
(b) $BC \to H$
(c) $BC \to E$
(d) $BC \to G$
(e) $BE \to G$
(f) $BE \to H$.

Step 3: Try to eliminate each FD.

(a) $ABF^+_{S2-(a)} = ABF$. We need this FD.

(b) $BC^+_{S2-(b)} = BCEGH$. We can remove this FD.

(c) $BC^+_{S2-\{(b),(c)\}} = BCG$. We need this FD.

(d) $BC^+_{S2-\{(b),(d)\}} = BCEGH$. We can remove this FD.

(e) $BE^+_{S2-\{(b),(d),(e)\}} = BEH$. We need this FD.

(f) $BE^+_{S2-\{(b),(d),(f)\}} = BEG$. We need this FD.

Our final set of FDs is:

(a) $ABF \to G$
(b) $BC \to E$
(c) $BE \to G$
(d) $BE \to H$.

Database Design

- To improve a badly-designed schema $R(A_1, \ldots A_n)$, we will decompose it into two smaller relations

$$R1(B_1, \ldots B_j) = \pi_{B1, \ldots Bj}(R)$$

$$R2(C_1, \ldots C_k) = \pi_{C1, \ldots Ck}(R)$$

such that:

$$\{B_1, \ldots B_j\} \cup \{C_1, \ldots C_k\} = \{A_1, \ldots A_n\}$$

$$R1 \bowtie R2 = R$$

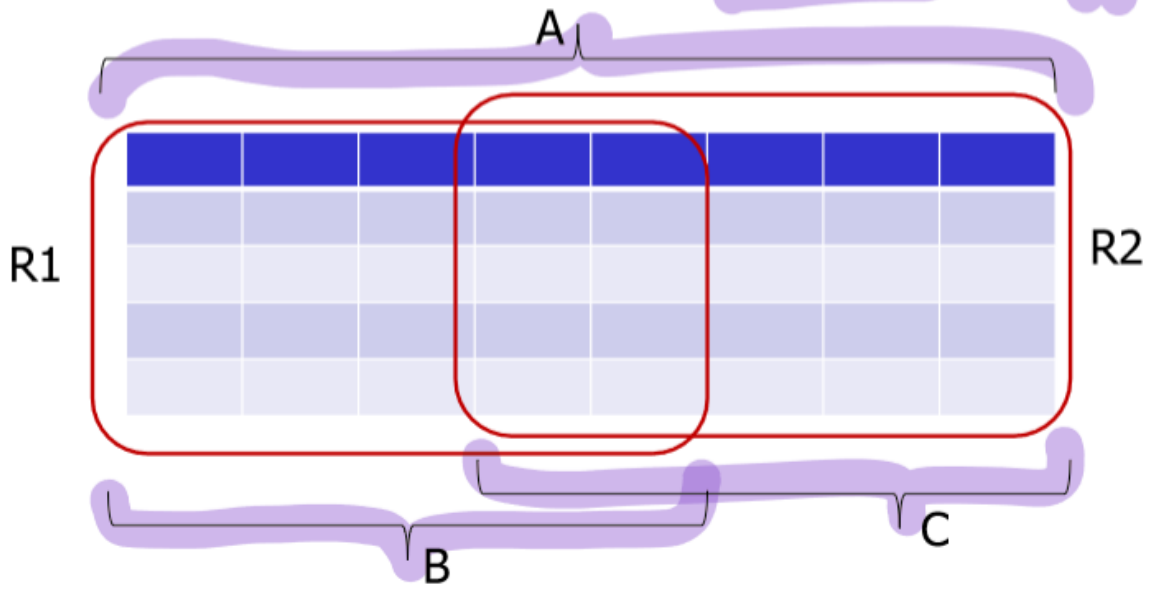
R(A1, ..., An) Set of attributes: A

Decompose into:

R1(B1, ..., Bk) = $\pi_{B1, \ldots Bk}(R)$ Set of attributes: B

R2(C1, ..., Cm) = $\pi_{C1, \ldots Ck}(R)$ Set of attributes: C

$B \cup C = A$ $R1 \bowtie R2 = R$



37

Boyce-Codd Normal Form

- we say a relation R is in **BCNF** if
  - for every nontrivial FD X → Y that holds in R, X is a superkey
  - **nontrivial** - Y is **not** contained in X
  - **superkey** - all attributes
- only things that functionally determine **everything** can functionally determine **anything**

R is a relation; F is a set of FDs.
Return the BCNF decomposition of R, given these FDs.

BCNF_decomp(*R*, *F*):

If an FD $X \rightarrow Y$ in F violates BCNF

Compute X^+.

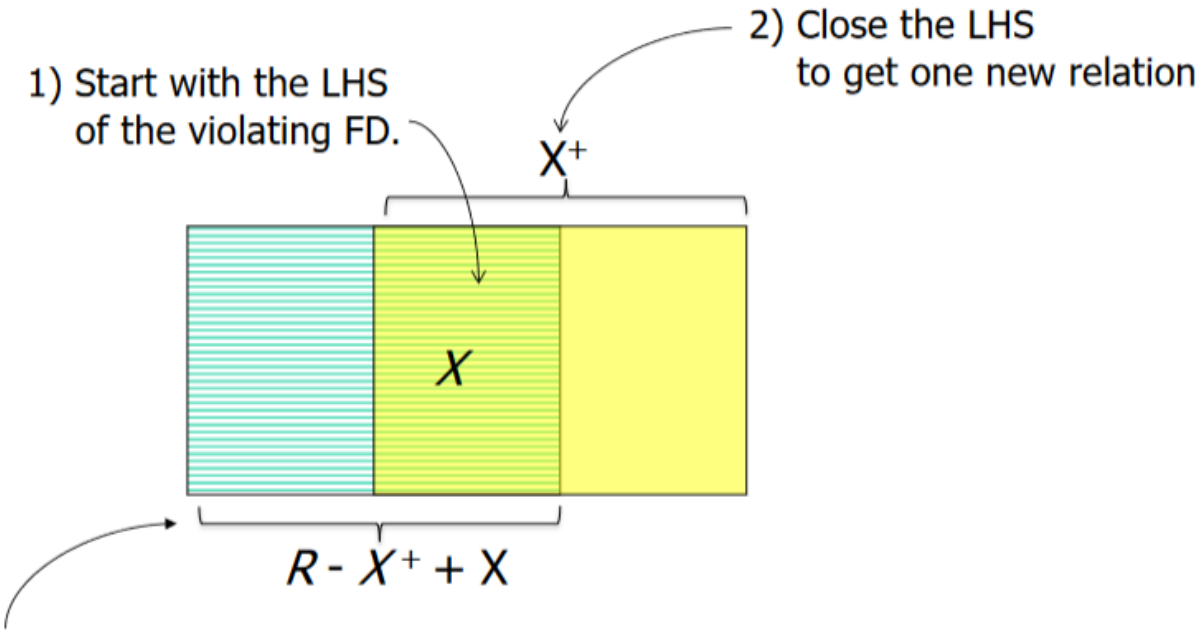
Replace R by two relations with schemas:

$R_1 = X^+$

$R_2 = R - (X^+ - X)$

Project the FD's F onto R_1 and R_2.

Recursively decompose R_1 and R_2 into BCNF.

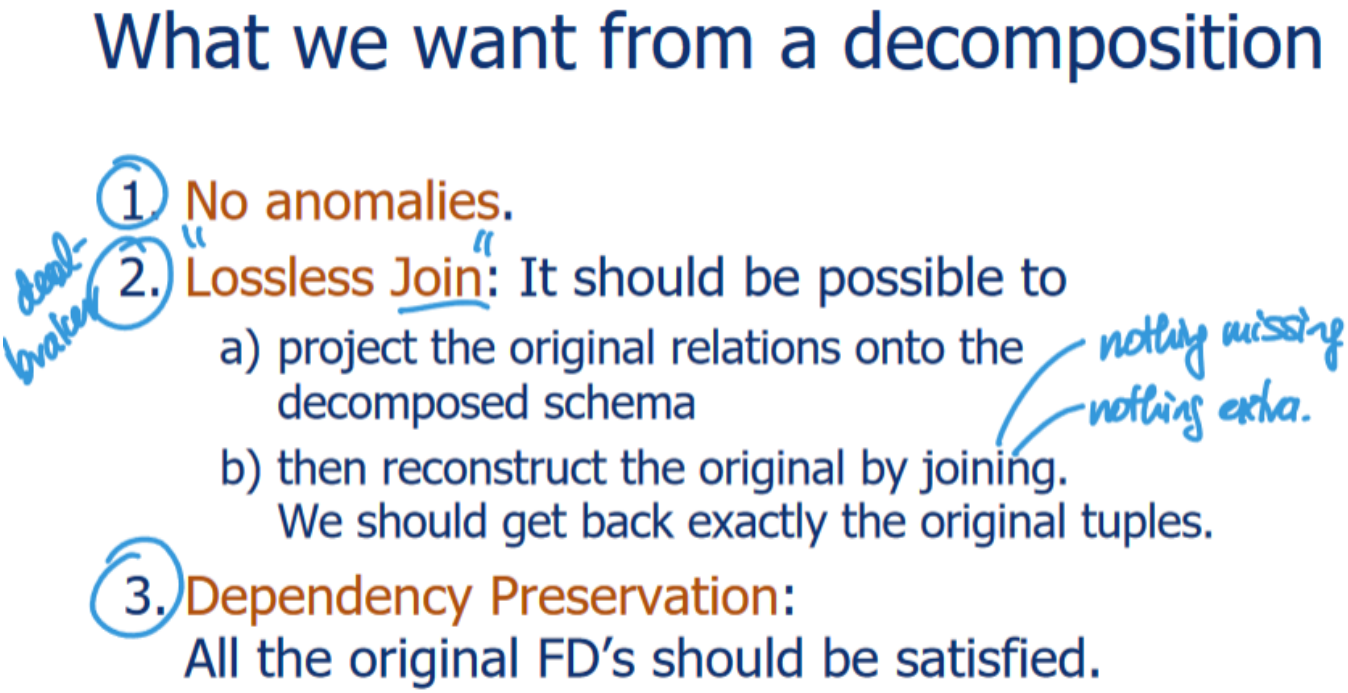


3) Everything except the new stuff is the other new relation.
X is in both new relations to make a connection between them.

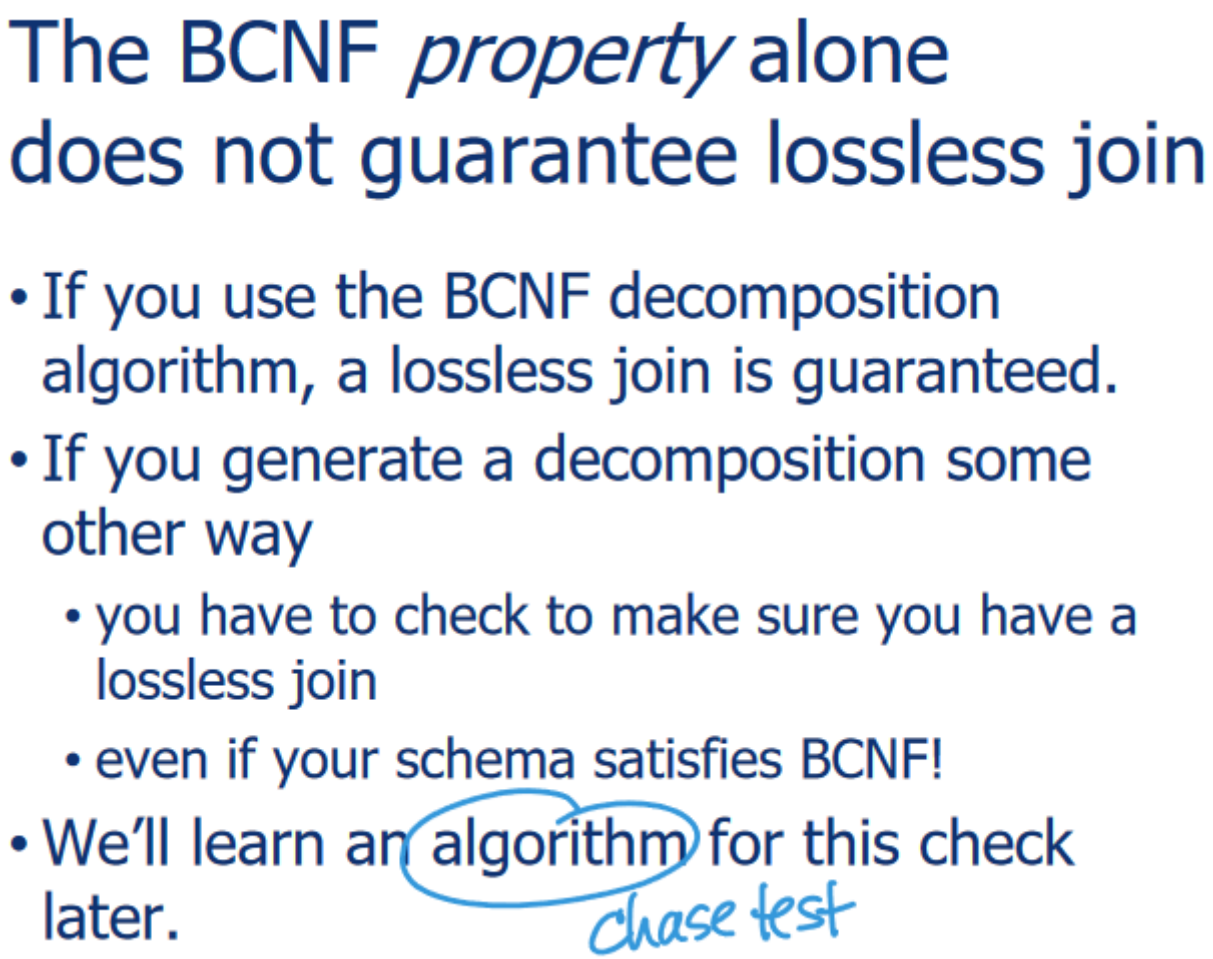
- don't need to know any keys → only superkeys matter
- don't need to know all superkeys
  - only need to check whether the LHS of each FD is a superkey

- when projecting FDs onto a new relation, if the new FD violate BCNF → abort the projection
- it is **impossible** or a 2-attribute relation violate BCNF

properties



- lossy join
  - for any decomposition → $r \subseteq r_1 \bowtie \ldots \bowtie r_n$
  - but should not $r \supseteq r_1 \bowtie \ldots \bowtie r_n$
- what BCNF offers → all but dependency preservation



3rd Normal Form

- an attribute is **prime** if it is a member of any key
- X → A violates 3NF iff
  - X is not a superkey **and** A is not prime

F is a set of FDs; L is a set of attributes.
Synthesize and return a schema in 3rd Normal Form.

3NF_synthesis(*F*, *L*):
  Construct a minimal basis M for F.
  For each FD $X \rightarrow Y$ in M
    Define a new relation with schema $X \cup Y$.
  If no relation is a superkey for *L*
    Add a relation whose schema is some key.

- offers all but no anomalies

chase test

- Start by assuming $t = abc\ldots$.
- For each i, there is a tuple s_i of R that has a, b, c,... in the attributes of R_i.
- s_i can have any values in other attributes.
- We'll use the same letter as in t, but with a subscript, for these components.

The algorithm

1. If two rows agree in the left side of a FD, make their right sides agree too.
2. Always replace a subscripted symbol by the corresponding unsubscripted one, if possible.
3. If we ever get a completely unsubscripted row, we know any tuple in the project-join is in the original (*i.e.,* the join is lossless).
4. Otherwise, the final tableau is a counterexample (*i.e.,* the join is lossy).

Suppose we decompose into relations NF, NL and NCG. Use the Chase Test to determine whether this is a lossless-join decomposition.

Solution: The chase test demonstrates that it is a lossless-join decomposition.
We start with:

N	F	L	C	G
n	f	l	2	3
n	4	l	5	6
n	7	8	**c**	**g**

Then, because $N \rightarrow FL$, we make these changes:

N	F	L	C	G
n	**f**	~~l~~	2	3
n	~~4~~ f	l	5	6
n	~~7~~ f	~~8~~ l	**c**	**g**

We observe that the tuple $< n, f, l, c, g >$ does occur. The Chase Test has succeeded.

Entity-Relationship Modelling and Practical Design

Entity-Relationship Model

basic concept

- entities
- relationships among them
- attributes describing the entities and the relationships

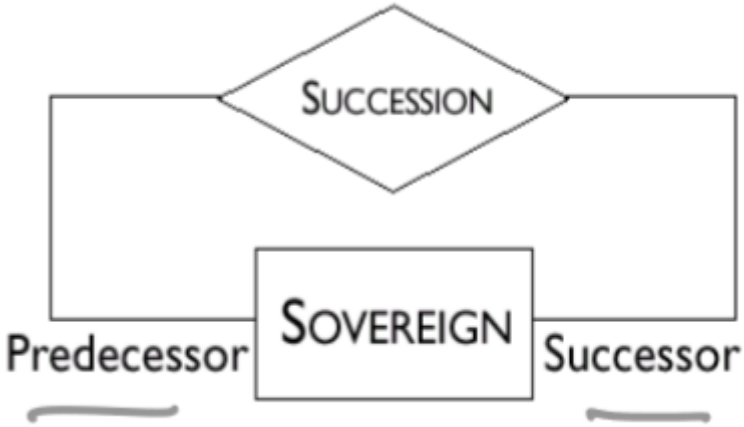
Instance	Schema
Entity, with attributes	Entity Set, with attributes
Relationship, with attributes	Relationship Set, with attributes

entity set

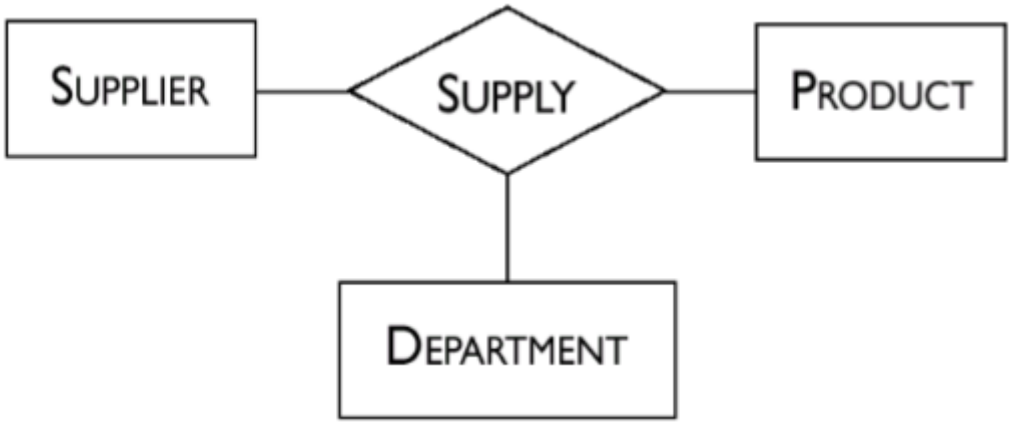
- represents a **category** of objects that have properties in common and an autonomous existence
  - city, department, employee
- **entity** - an instance of an entity set
  - Peterson is an employee

relationship set

- an association between **2+** entity sets
  - residence is a relationship set between entity sets City and Employee
- **relationship** - an instance of a n-ary relationship set
  - the pair is an instance of a relationship residence
- recursive relationships
  - relate an entity set to itself
  - asymmetric → indicates the two **roles** that the entity plays in the relationship

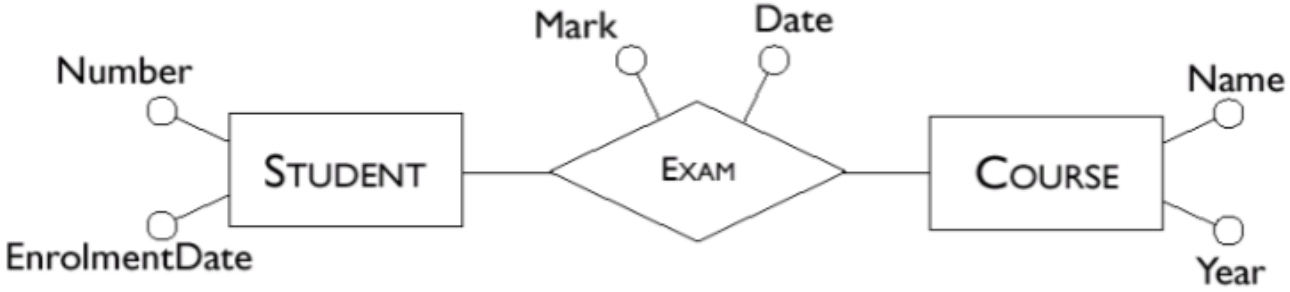


- ternary relationship

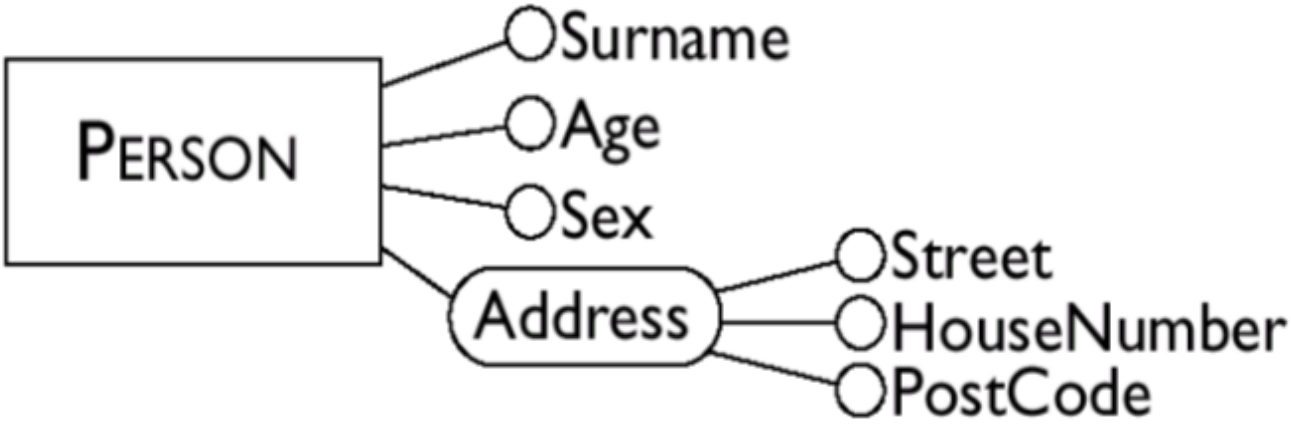


attributes

- describe elementary properties of entities or relationships (surname, salary, age)
- may be **single-valued**, or **multi-valued**



- composite attribute
  - grouped attributes of the same entity or relationship that have closely connected meaning or uses

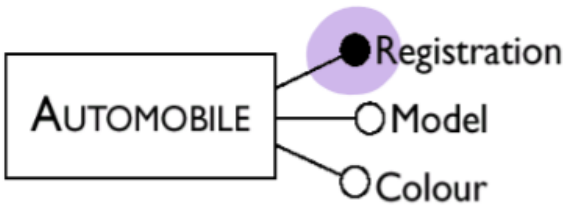


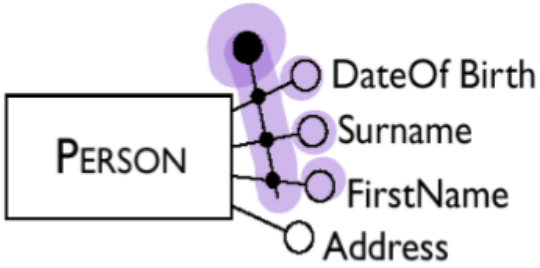
keys

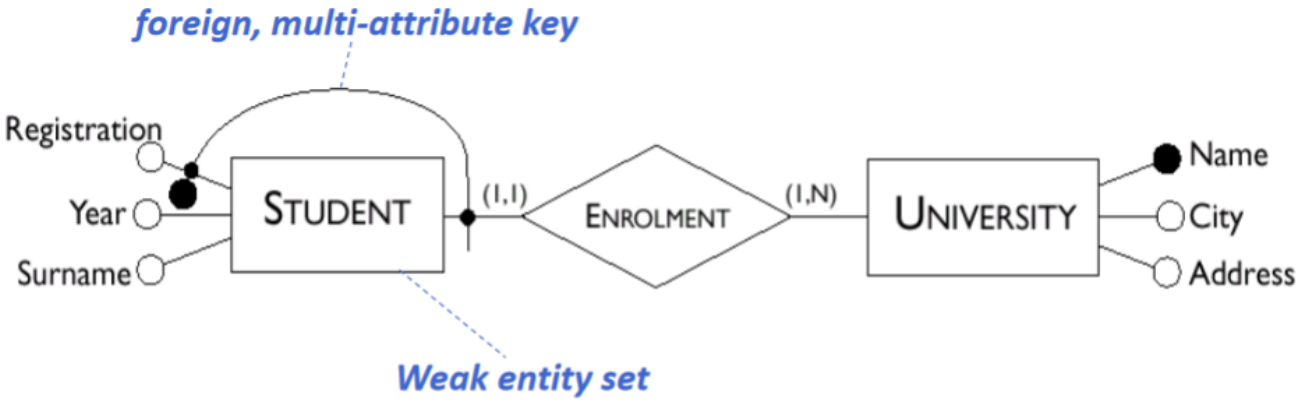
- solid circle
- if **multi-attribute**, connect with a line
- formed by one or more attributes of the **entity itself** → **internal** key
  - an entity set **doesn't** have a key among its attributes → **weak entity set**
  - the keys of related entities are brought in → **foreign keys**
- the key for a relationship set consists of the keys of the **entity set** that it relates

requirement

- each **attribute** in a key **must** have **(1,1)** cardinality
- a foreign key for a weak entity set must come through a relationship which the entity set participates in with cardinality (1,1)
- a foreign key may involve an entity that has itself a foreign key, as long as cycles are not generated
- each entity set **must** have **at least one** key



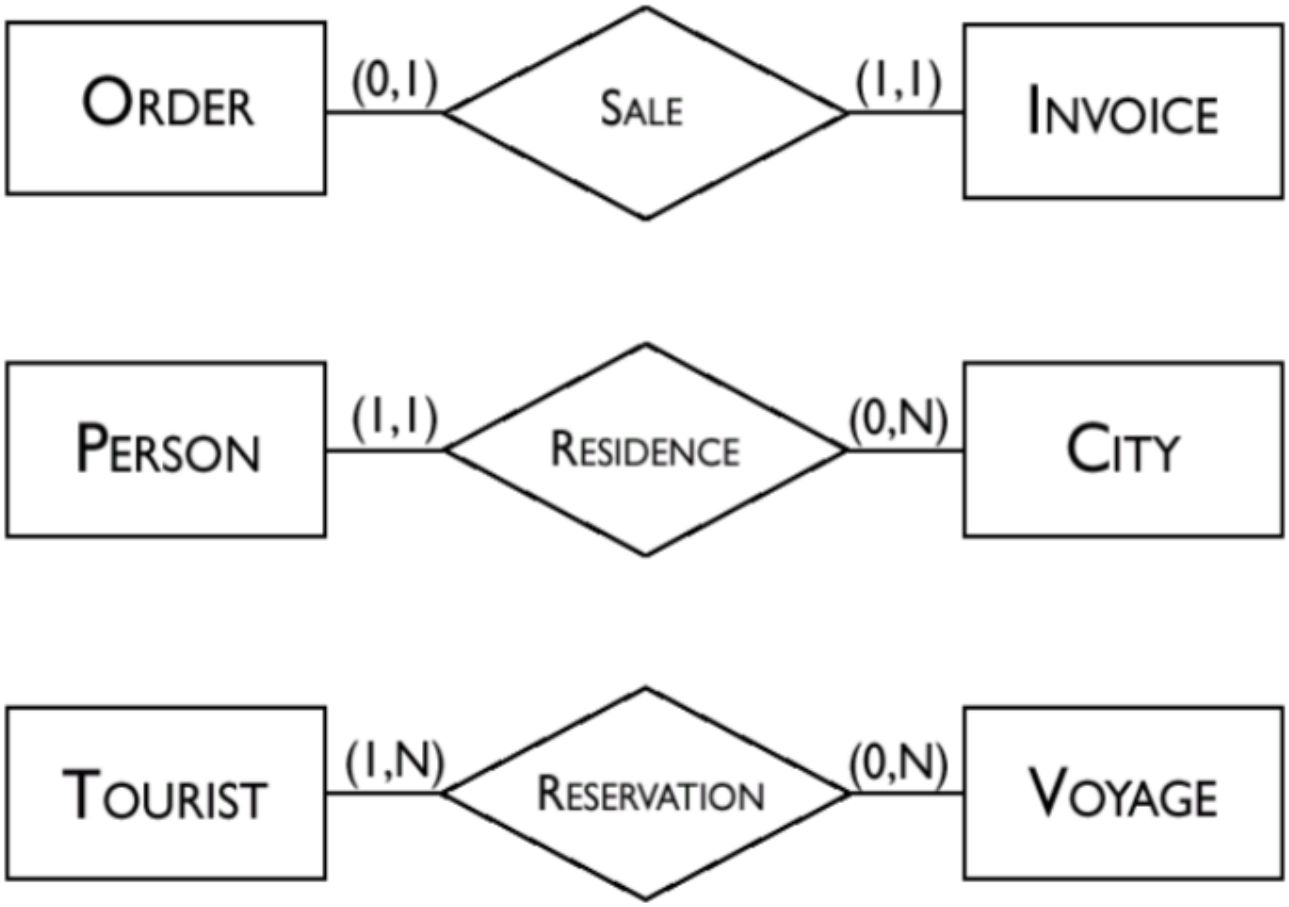




cardinality

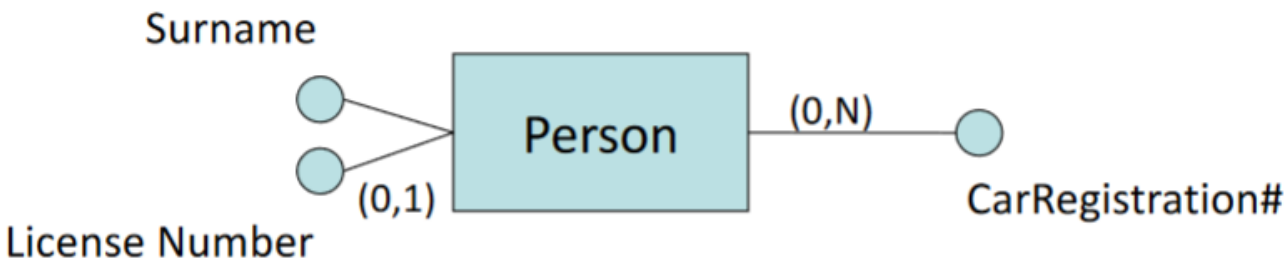
relationship

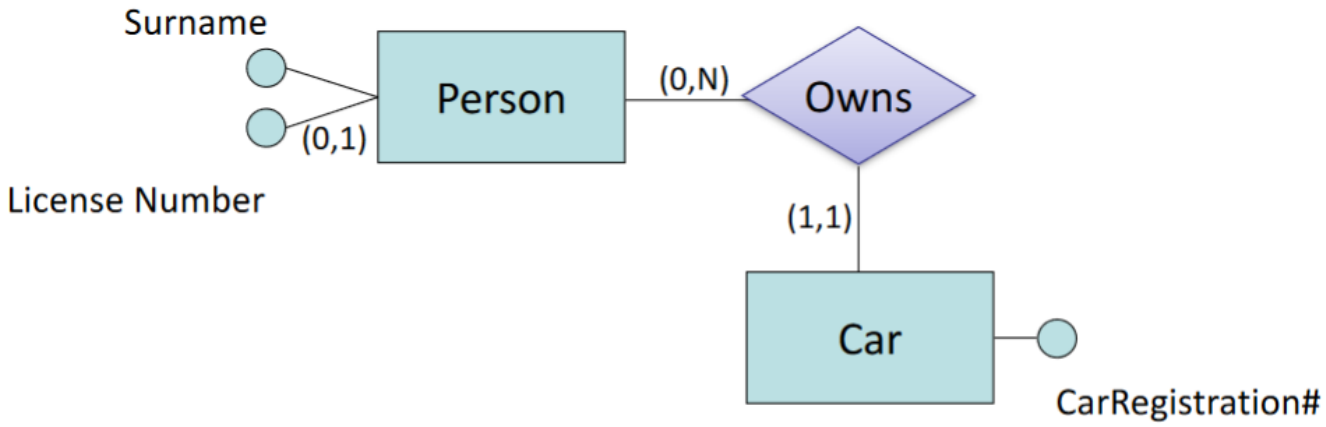
- each entity set participates in a relationship set with a **min** and **max** cardinality
- **min <= max**
- min
  - if 0, participation is **optional**
  - if 1, participation is **mandatory**
- max
  - if 1, each instance of the entity is associated **at most with a single** instance of the relationship
  - if >1, **multiple**
  - write **N** to indicate no upper limit



attribute

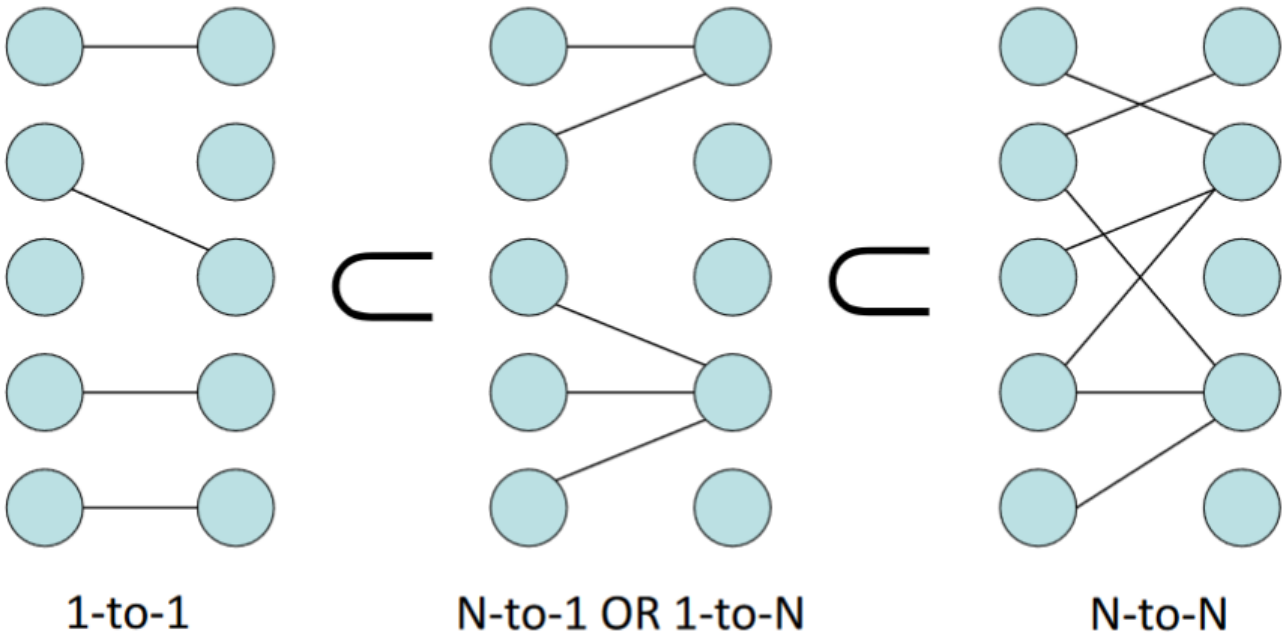
- describe min/max number of values an attribute can have
- (1,1) → **single-value attribute** (can be omitted)
- may also be null, or have several values → **multi-valued attribute**
  - represent situations that can be modeled with additional entities





multiplicity of relationship

- suppose entity sets E1 and E2 participate in relationship R with cardinalities **(n1,N1)** and **(n2, N2)**
- we say that the multiplicity of R is **N1-to-N2** or **N2-to-N1**
- less information than (min, max) notation



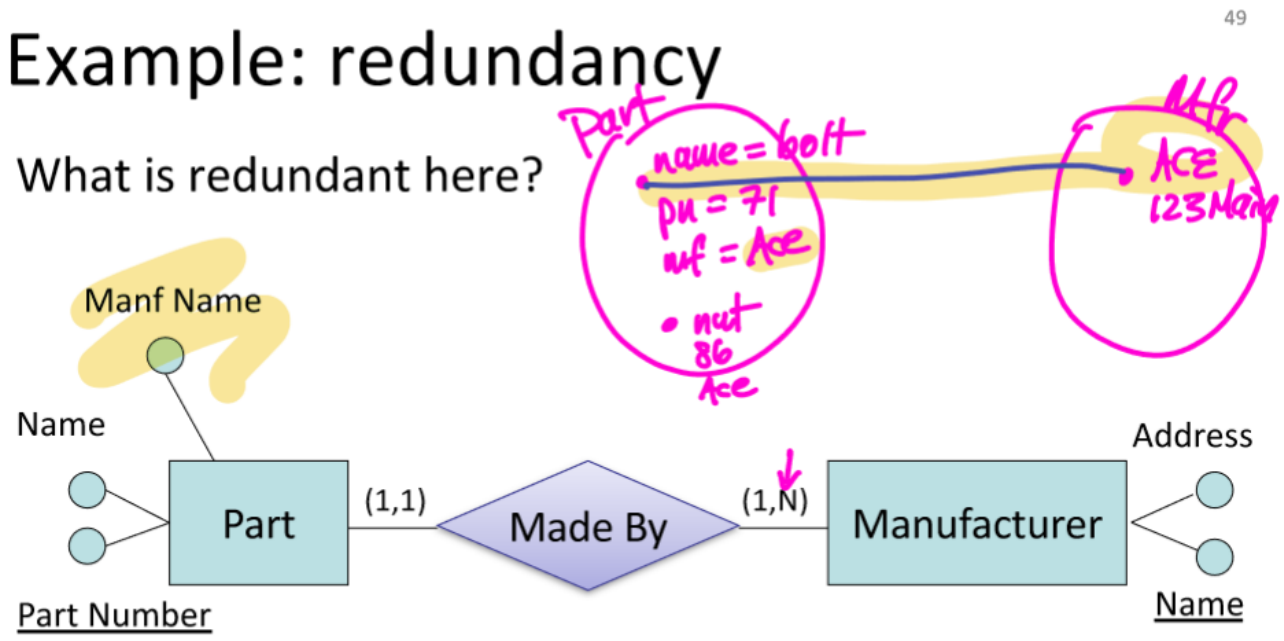
From Model To DB Schema

restructuring

It includes (not necessarily in this order):

a. Analysis of redundancies
b. Choosing entity set vs attribute
c. Limiting the use of weak entity sets
d. Settling on keys
e. Creating entity sets to replace attributes with cardinality greater than one

analysis of redundancies



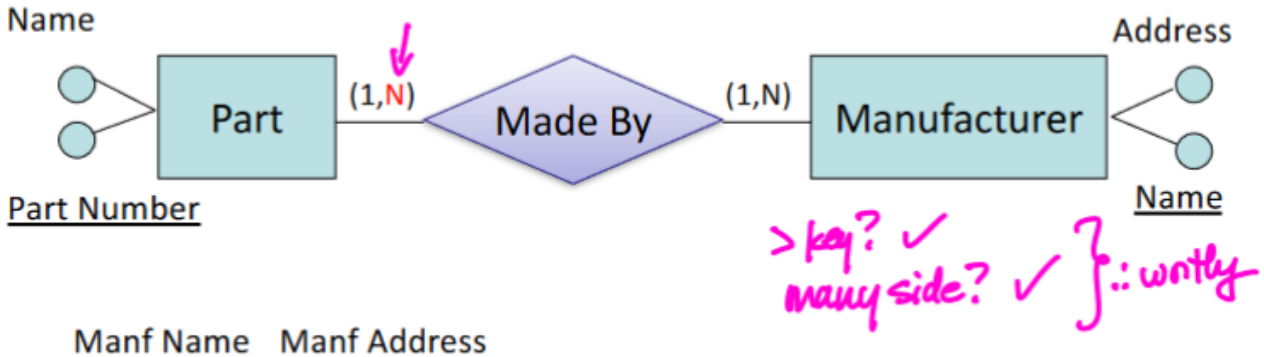
entity vs attribute

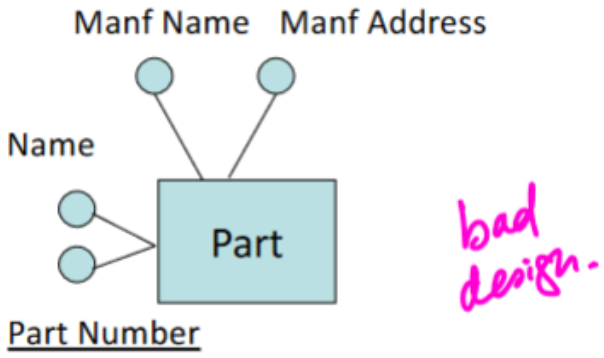
- an entity set should satisfy at least one of
  - it is **more** than the key of something; it has **at least one** non-key attribute, or
  - it is the **many** in a many-one or many-many relationship (avoid redundancy)

- a **thing** in its own right → entity set
- a **detail** about some other thing → attribute

E.S. vs. attributes: examples

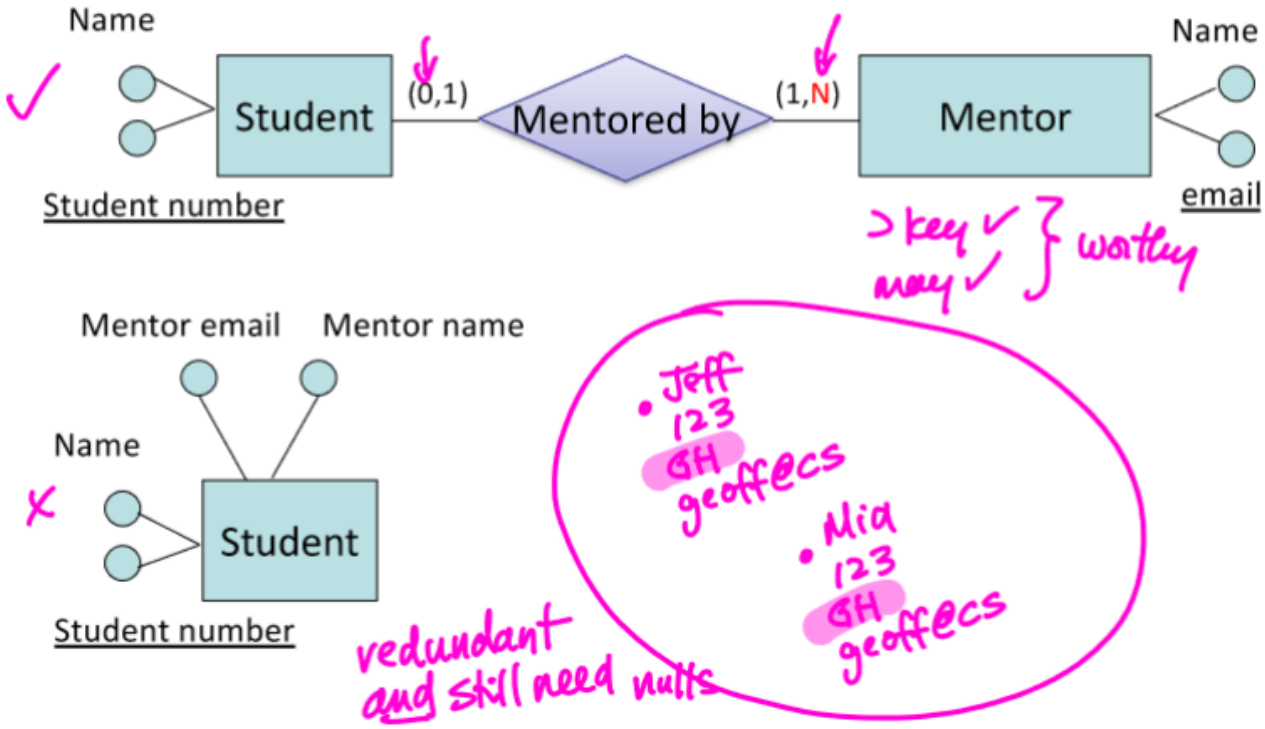
Domain fact change: A part can have more than one manufacturer ...





E.S. vs. attributes: examples

Domain fact change: A mentor can have more than one mentee ...



weak entity sets

- **never** use weak entity sets

settling on keys

- make sure that **every entity set has a key**

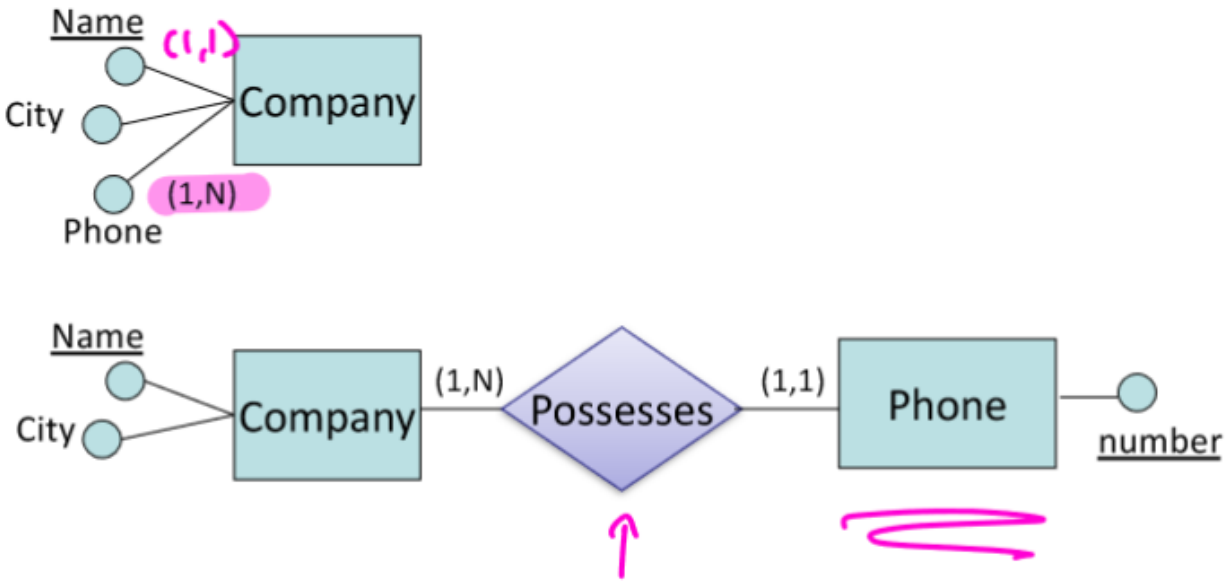
Keep in mind:

- Attributes with null values cannot be part of primary keys
- Internal keys are preferable to external ones
- A key that is used by many operations to access instances of an entity is preferable to others
- A key with one/few attributes is preferrable
- An integer key is preferrable

- **avoid** multi-attribute and string keys
  - wasteful, integer id can save space and a lot of typing
  - break encapsulation → security/privacy hole
  - nasty interaction of above two point → internal id always exist, are **immutable, unique**

Attributes with cardinality > 1

The relational model doesn't allow multi-valued attributes. We must convert these to entity sets.



translating

- Starting from an E/R schema, an equivalent relational schema is constructed
  - "equivalent": a schema capable of representing the same information
- A good translation should also:
  - not allow redundancy
  - not invite unnecessary null values

- entity set becomes a relation
  - attribute → **attributes** of the entity set
- relationship becomes a relation
  - attribute → **keys** of entity sets that it connects + attributes of itself

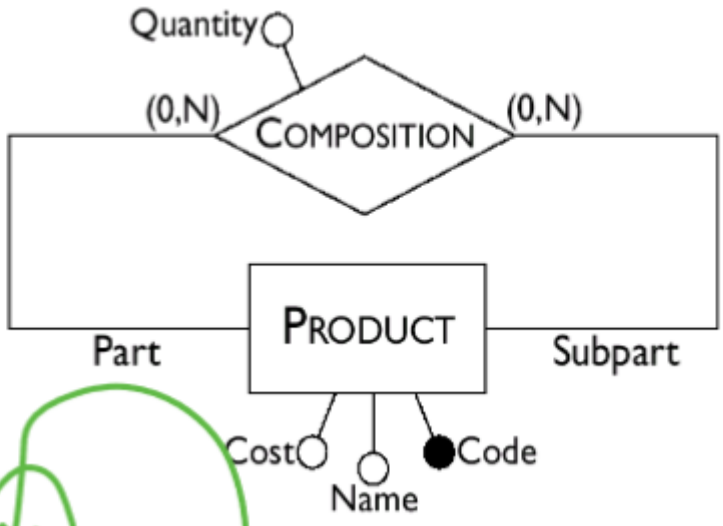
Employee(<u>Number</u>, Surname, Salary)

Project(<u>Code</u>, Name, Budget)

Participation(<u>Number, Code</u>, StartDate)

Participation[Number] ⊆ Employee[Number]

Participation[Code] ⊆ Project[Code]

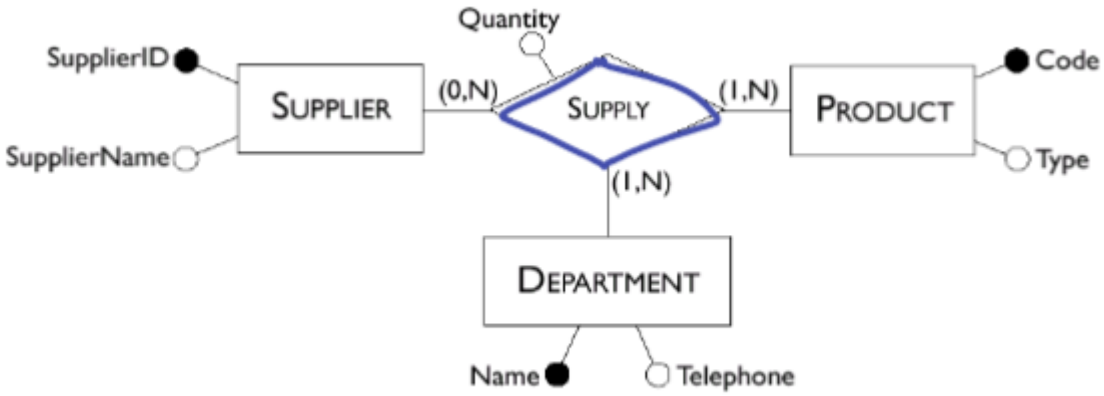


✓Product(Code, Name, Cost)

Composition(Part, SubPart, Quantity)

Composition[Part] ⊆ Product[Code]

Composition[SubPart] ⊆ Product[Code]



✓Supplier(SupplierID, SupplierName)

✓Product(Code, Type)

✓Department(Name, Telephone)

Supply(Supplier, Product, Department, Quantity)

Supply[Supplier] ⊆ Supplier[SupplierID]

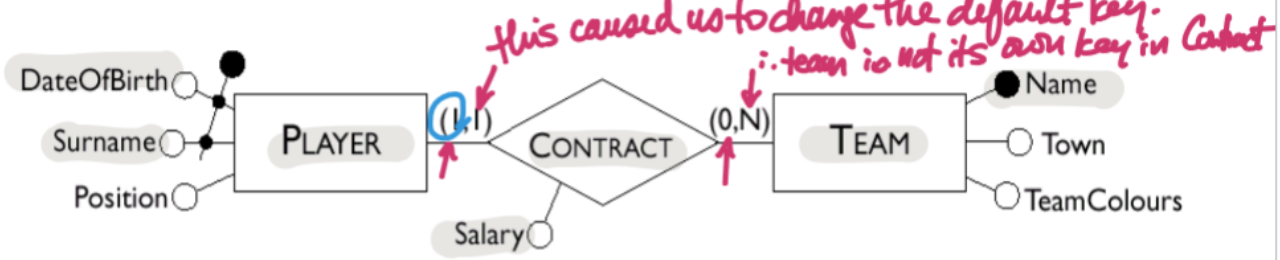
Supply[Product] ⊆ Product[Code]

Supply[Department] ⊆ Department[Name]

simplify

One-to-Many Relationships

with mandatory participation on the "one" side



Standard translation:

circular

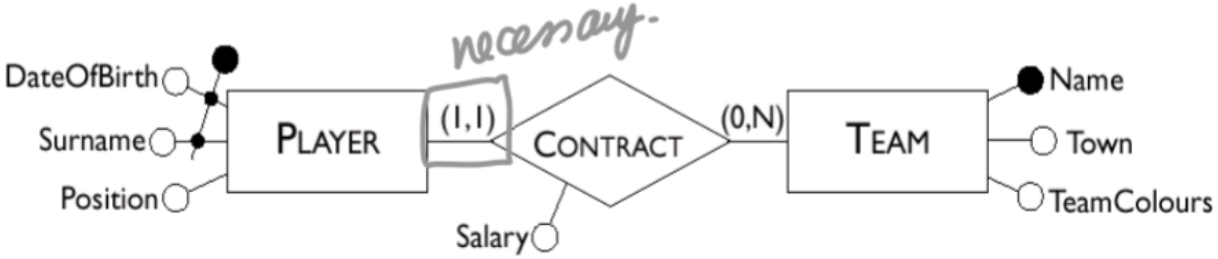
Player(Surname, DOB, Position)
Team(Name, Town, TeamColours)
Contract(PlayerSurname, PlayerDOB, Team, Salary)

Contract[PlayerSurname, PlayerDOB]⊆Player[Surname, DOB]

Contract[Team] ⊆Team[Name]

One-to-Many Relationships

with mandatory participation on the "one" side



Simpler translation:

put the contract info into Player

Player(Surname, DOB, Position, TeamName, Salary)
Team(Name, Town, TeamColours)

Player[TeamName] ⊆Team[Name]

One-to-One Relationships

with mandatory participation for both

Simpler v1 (with Management info moved over to Head):

Head(<u>Number</u>, Name, Salary, Department, StartDate)

Department(<u>Name</u>, Telephone, Branch)

Head[Department] ⊆ Department[Name]

Simpler v2 (with Management info moved over to Department):

Head(<u>Number</u>, Name, Salary)

Department(<u>Name</u>, Telephone, Branch, HeadNumber, StartDate)

Department[HeadNumber] ⊆ Head[Number]

One-to-One Relationships

with optional participation for one

Simpler:

Employee(<u>Number</u>, Name, Salary)

Department(<u>Name</u>, Telephone, Branch, Head, StartDate)

Department[Head] ⊆ Employee[Number]

- Express the foreign-key constraints
- Consider better names for the referring attributes
- Express the ”max 1” constraints
- Express the “min 1” constraints

What about “min 1” constraints?

In our translation from E/R to a relational schema, we expressed the constraints using relational notation.

But in SQL, only some of them can be written as foreign keys.

- They must refer to attributes that are primary key or unique.

Indice

With a database there are some new realities

This leads to some important design goals:

- Organize disk pages so we can *use* the whole page we had to read
- To minimize pointer following, use a huge branching factor. A very wide tree will be much shorter.

A B-tree is a data structure that achieves these goals

DBMSs typically use them to provide very fast access by primary key

```
csc343h-dianeh=> \d offering

     Table "university.offering"
    Column    |         Type          | Modifiers
--------------+-----------------------+-----------
 oid          | integer               | not null
 cnum         | integer               |
 dept         | department            |
 term         | integer               | not null
 instructor   | character varying(40) |

Indexes:
    "offering_pkey" PRIMARY KEY, btree (oid)
```

What a B-tree looks like

here, just showing keys (not the rest...)