

# Databases and Information Systems

## SQL

## Functional Dependencies & Decomposition

Dmitriy Traytel

slides mostly by Marcos Vaz Salles



# Null Values

- Field values in a tuple are sometimes **unknown**
  - e.g., a rating has not been assigned
- Field values are sometimes **inapplicable**
  - e.g., no spouse's name
- SQL provides a special value **null** for such situations.

# Queries and Null Values

- What if S.Age is NULL?
  - S.Age > 25 returns NULL!
- Implies a predicate can return 3 values
  - True, false, NULL
  - **Three-valued logic!**
- Where clause eliminates rows that do not return true (i.e., that are false or NULL)

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	NULL
58	rusty	10	35.0

SELECT	sname
FROM	Sailors
WHERE	age > 25

sname
dustin
rusty

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	NULL
58	rusty	10	35.0

# Three-valued Logic

```
SELECT    sname
FROM      Sailors
WHERE     NOT(age > 25) OR rating > 7
```

sname
rusty
lubber

- What if one or both of S.age and S.rating are NULL?

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
NULL	TRUE	NULL	TRUE	NULL
NULL	NULL	NULL	NULL	NULL
NULL	FALSE	FALSE	NULL	NULL
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Sailors

# Expressions and Strings

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	NULL
58	rusty	10	35.0

```
SELECT age, age-5 AS age5, 2*age AS age2
FROM Sailors
WHERE sname LIKE '_u%'
```

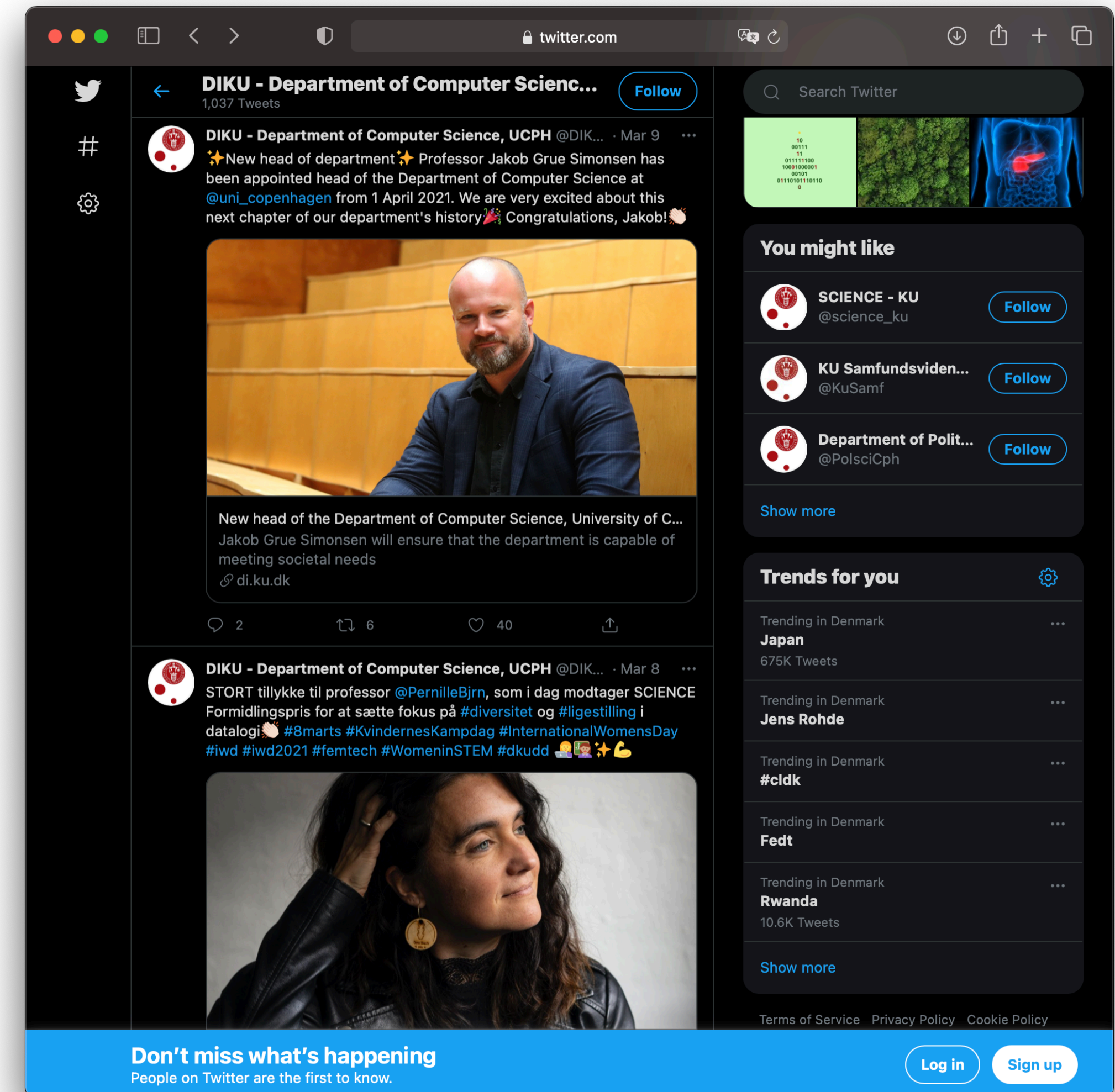
age	age5	age2
45	40	90
35	30	70
NULL	NULL	NULL

- Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names' second letter is u
- **AS** is used to name fields in result
- **LIKE** is used for string matching
  - **\_** matches a single arbitrary character
  - **%** matches for 0 or more arbitrary characters.



# Discussion: Querying for NULLs

- We model Twitter using the following three tables:
  - Users(uid, name, joineddate)
  - Posts(pid, uid, date, text)
  - Mentions(pid, uid)
- Formulate in SQL: List the names of users who were mentioned in a post, but whose joined date is unknown



# Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of **relation-list**
  - Discard resulting tuples if they fail **condition**
  - Delete attributes that are not in **target-list**
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query!
  - An optimizer will find more efficient strategies to compute the same answers.

# Find sid's of sailors who've reserved a red and a green boat

Key field!

```
SELECT S.sid
FROM   Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
      AND B.color='red'
INTERSECT
SELECT S.sid
FROM   Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
      AND B.color='green'
```

- We can build queries with **nested queries!**
  - In SELECT, FROM, WHERE clauses or with set operations



Find sid's of sailors who have **not** reserved boat #103

```
SELECT S.sid
FROM   Sailors S
      LEFT OUTER JOIN
      (SELECT sid
       FROM   Sailors NATURAL JOIN Reserves R
       WHERE  R.bid = 103) AS S_103
      ON S.sid = S_103.sid
WHERE  S_103.sid IS NULL
```


# Common Table Expressions (WITH)

```
WITH S_103(sid) AS (  
    SELECT sid  
    FROM    Sailors NATURAL JOIN Reserves  
    WHERE   bid = 103  
)  
SELECT S.sid  
FROM    Sailors S  
        LEFT OUTER JOIN  
        S_103  
        ON S.sid = S_103.sid  
WHERE S_103.sid IS NULL
```

# Nested Queries (with Correlation)

- Find names of sailors who have reserved boat #103:

```
SELECT  S.sname
FROM    Sailors S
WHERE   EXISTS (SELECT *
                  FROM    Reserves R
                  WHERE    R.bid=103 AND S.sid=R.sid)
```



# Nested Queries (with Correlation)

- Find names of sailors who have **not** reserved boat #103:

```
SELECT  S.sname
FROM    Sailors S
WHERE   NOT EXISTS (SELECT *
                    FROM    Reserves R
                    WHERE   R.bid=103 AND S.sid=R.sid)
```



# Different ways to say the same thing

```
SELECT S.sid
FROM   Sailors S
WHERE  NOT EXISTS (SELECT *
                   FROM   Reserves R
                   WHERE  R.sid = S.sid AND R.bid = 103)
```

```
SELECT S.sid
FROM   Sailors S
      LEFT OUTER JOIN
      (SELECT S2.sid
       FROM   Sailors S2, Reserves R
       WHERE  S2.sid = R.sid
              AND  R.bid = 103) AS S3
      ON S.sid = S3.sid
WHERE  S3.sid IS NULL
```

```
SELECT S.sid
FROM   Sailors S
      EXCEPT
      SELECT S.sid
      FROM   Sailors S,
            Reserves R
      WHERE  S.sid = R.sid
            AND  R.bid = 103
```



# And How About These?

R		S	
a	b	b	c
1	2	2	5
3	4	2	6

```
SELECT R.a
FROM R, S
WHERE R.b = S.b
```

a
1
1

```
SELECT a
FROM R
WHERE b IN (SELECT b FROM S)
```

a
1

# Set-Comparison Operators

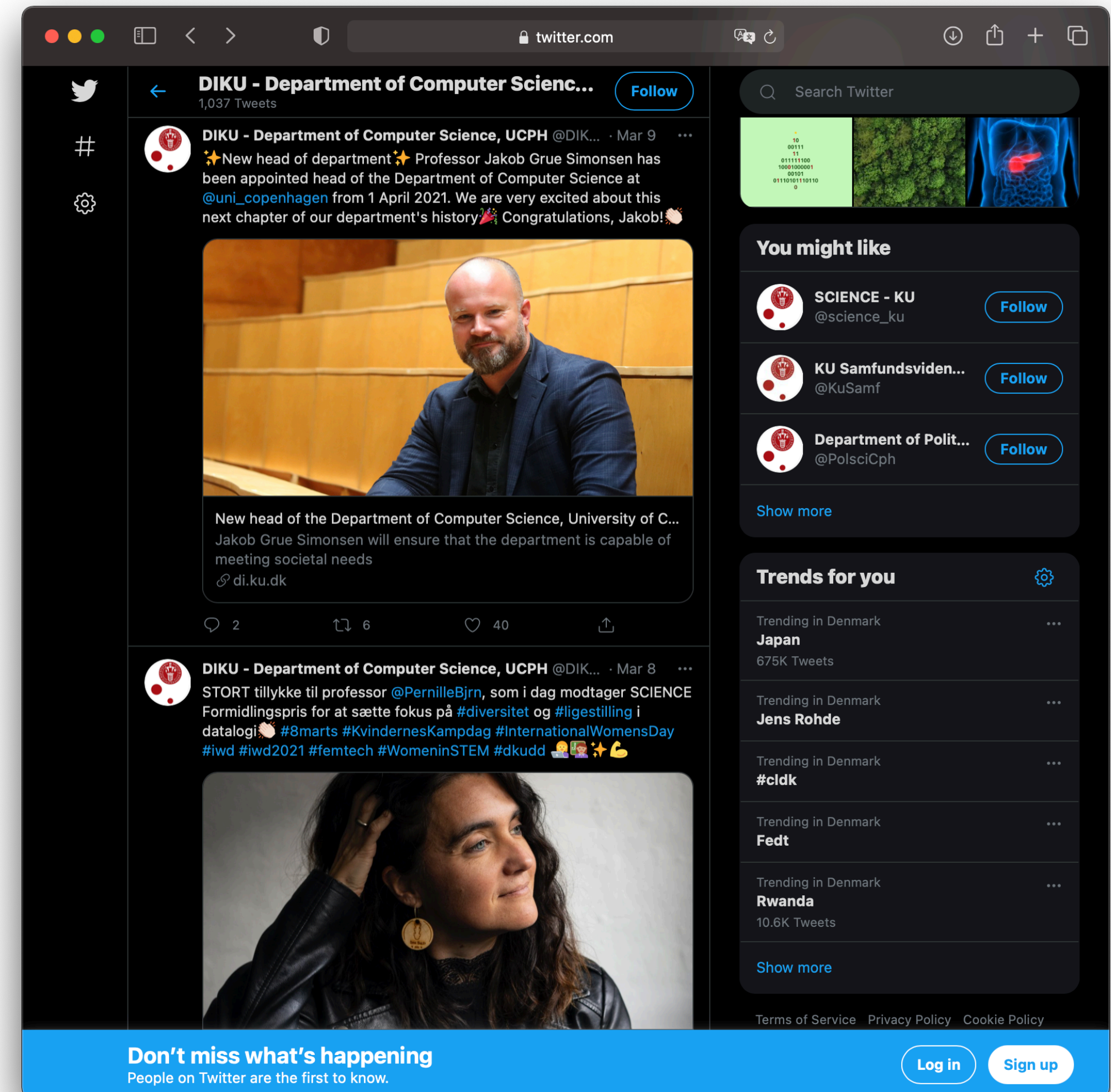
- `op ANY`, `op ALL`
- `op` can be `>`, `<`, `=`, `<=`, `>=`, `<>`
- Find sailors whose rating is greater than that of all sailors called lubber:

```
SELECT *  
FROM   Sailors S  
WHERE  S.rating > ALL (SELECT S2.rating  
                        FROM   Sailors S2  
                        WHERE  S2.sname='lubber')
```

# Recap: Querying in SQL

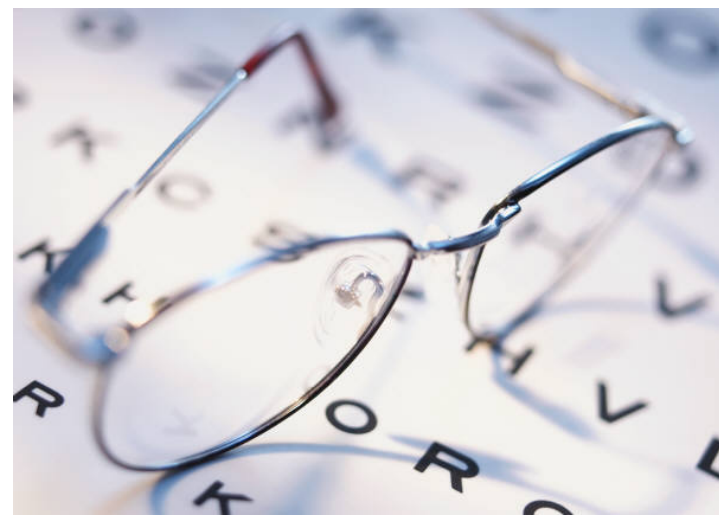
- We model Twitter using the following three tables:
  - Users(uid, name, joineddate)
  - Posts(pid, uid, date, text)
  - Mentions(pid, uid)
- Formulate in SQL:
  - Find the names of the users who have not posted anything, but have been mentioned by others.

```
SELECT name
FROM Users NATURAL JOIN
(SELECT DISTINCT uid FROM Mentions) AS X
WHERE uid NOT IN (SELECT uid FROM Posts)
```





# What should we learn today?



- Explain the semantics of **aggregation** operations and **grouping** in SQL
- Formulate queries that combine **joins**, **sub-queries**, **grouping**, and **aggregation**
- Define and explain the notion of **functional dependencies** (FDs), and apply rules to reason about FDs including Armstrong's Axioms
- Review the notion of **keys** and how to **determine** them by reasoning on functional dependencies
- Explain the issues in **decomposition** of relations and verify whether decompositions are **lossless-join** and **dependency-preserving**
- Apply the **chase test** to determine if a decomposition is lossless-join

# Aggregate Operators

- Significant extension of relational algebra

COUNT (\*)  
COUNT ( [DISTINCT] A)  
SUM ( [DISTINCT] A)  
AVG ( [DISTINCT] A)  
MAX (A)  
MIN (A)

*single column*

```
SELECT COUNT (*)  
FROM   Sailors S
```

```
SELECT AVG (S.age)  
FROM   Sailors S  
WHERE  S.rating = 10
```

```
SELECT COUNT(DISTINCT S.rating)  
FROM   Sailors S  
WHERE  S.sname = 'lubber'
```



Find name and age of the oldest sailor(s)  
with rating > 7

```
SELECT  S.sname, S.age
FROM    Sailors S
WHERE   S.rating > 7 AND
        S.age = (SELECT  MAX(S2.age)
                  FROM    Sailors S2
                  WHERE   S2.rating > 7)
```

# Aggregate Operators

- So far, we've applied aggregate operators to all (qualifying) tuples
- Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: *Find the age of the youngest sailor for each rating level.*
  - If rating values go from 1 to 10; we can write 10 queries that look like this:

For  $i = 1, 2, \dots, 10$ :

```
SELECT MIN(S.age)
FROM   Sailors S
WHERE  S.rating = i
```

# GROUP BY

```
SELECT      [DISTINCT] target-list  
FROM        relation-list  
[WHERE      condition]  
GROUP BY    grouping-list
```

Find the age of the youngest sailor for each rating level

```
SELECT      S.rating, MIN(S.Age)  
FROM        Sailors S  
GROUP BY    S.rating
```

# Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of **relation-list**
  - Discard resulting tuples if they fail **condition**
  - **Remaining tuples are partitioned into groups by the value of the attributes in grouping-list**
  - **One answer tuple is generated per group**
  - Delete attributes that are not in **target-list**
- Recall: Does not imply query will actually be evaluated this way!

Find the age of the youngest sailor with age  $\geq 18$ ,  
for each rating with at least one such sailor

```
SELECT  S.rating, MIN(S.age)
FROM    Sailors S
WHERE   S.age >= 18
GROUP BY S.rating
```

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	15.5
71	zorba	10	16.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

sid	sname	rating	age
22	dustin	7	45.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

rating	
7	35.0
1	33.0
10	35.0



# Are These Queries Correct?

```
SELECT    MIN(S.Age)
FROM      Sailors S
GROUP BY  S.rating
```

```
SELECT    S.name, S.rating, MIN(S.Age)
FROM      Sailors S
GROUP BY  S.rating
```

# What does this query compute?

```
SELECT    B.bid, COUNT (*) AS scount
FROM      Reserves R, Boats B
WHERE     R.bid = B.bid AND B.color = 'red'
GROUP BY  B.bid
```

# GROUP BY and HAVING

```
SELECT    [DISTINCT] target-list
FROM      relation-list
[WHERE    qualification]
GROUP BY  grouping-list
HAVING    group-qualification
```

Find the age of the youngest sailor with age  $\geq 18$   
for each rating level with at least 2 such sailors

```
SELECT    S.rating, MIN(S.Age)
FROM      Sailors S
WHERE     S.age  $\geq$  18
GROUP BY  S.rating
HAVING    COUNT(*)  $>$  1
```

# Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of **relation-list**
  - Discard resulting tuples if they fail **condition**
  - Remaining tuples are partitioned into groups by the value of the attributes in **grouping-list**
  - **The *group-qualification* is applied to eliminate some groups**
  - One answer tuple is generated per group
  - Delete attributes that are not in **target-list**
- Recall: Does not imply query will actually be evaluated this way!

Find the age of the youngest sailor with age  $\geq 18$ ,  
for each rating with at least 2 such sailors

```
SELECT    S.rating, MIN(S.age)
FROM      Sailors S
WHERE     S.age >= 18
GROUP BY  S.rating
HAVING    COUNT(*) > 1
```

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	15.5
71	zorba	10	16.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

- Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes “*unnecessary*”
- 2nd column of result is unnamed (Use AS to name it)

sid	sname	rating	age
22	dustin	7	45.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

rating	
7	35.0



Find the age of the youngest sailor with age  $\geq 18$ ,  
for each rating with at least 2 sailors (of any age)

```
SELECT  S.rating, MIN(S.age)
FROM    Sailors S
WHERE   S.age >= 18
GROUP BY S.rating
HAVING  1 < (SELECT COUNT (*)
              FROM    Sailors S2
              WHERE   S.rating = S2.rating)
```

# List the sid's of all sailors along with their reservation count

```
SELECT sid,  
       SUM(CASE  
           WHEN R.bid IS NULL THEN 0  
           ELSE 1)  
FROM   Sailors  
       NATURAL LEFT OUTER JOIN  
       Reserves R  
GROUP BY sid
```

- Note: COUNT(\*) counts all rows; otherwise, aggregations over columns ignore NULL values and return NULL when aggregating the empty set

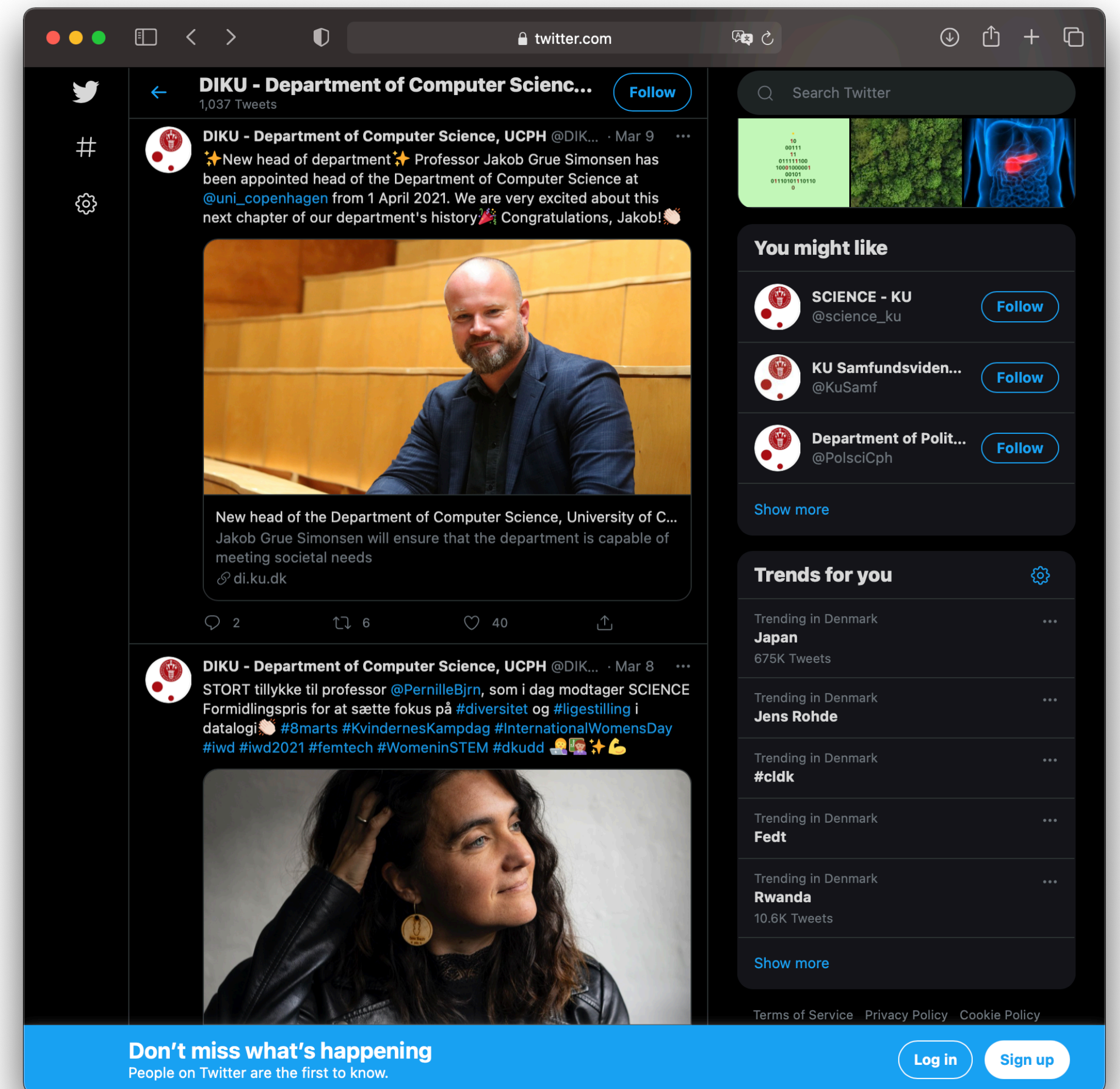
Find the average age for each rating, and order results in ascending order on average age

```
SELECT    S.rating, AVG(S.age) AS avgage
FROM      Sailors S
GROUP BY  S.rating
ORDER BY  avgage
```

# Discussion: Querying in SQL

- We model Twitter using the following three tables:
  - Users(uid, name, joineddate)
  - Posts(pid, uid, date, text)
  - Mentions(pid, uid)
- Formulate in SQL:
  - Find the names of the users whose posts taken together mention more than 1000 different people

```
SELECT name
FROM Users NATURAL JOIN
(SELECT P.uid AS uid
FROM Posts P INNER JOIN Mentions M USING (pid)
GROUP BY P.uid
HAVING COUNT(DISTINCT M.uid) > 1000) AS X
```





[illegible]

```

Mr. Dmitry Traytel

```

# Questions so far

# Last competition task: Art with SQL

Inspiration: [https://wiki.postgresql.org/wiki/Mandelbrot\\_set](https://wiki.postgresql.org/wiki/Mandelbrot_set)

## Evaluation criteria: creativity, beauty

Anonymized evaluation: submit your SQL script by email to [traytel@di.ku.dk](mailto:traytel@di.ku.dk)

# Did we get our DB schema right?

- Database design subjective: many schemas possible
- It would be good to know whether our schema is “good”
- What does “good” mean?
  - Does the schema capture all information with no or minimal redundancy?
  - Is it efficient to answer queries over the database?
  - Can we enforce all constraints?



# General Constraints

- Useful when more general ICs than keys are involved
- Can use queries to express constraint
- Constraints can be named

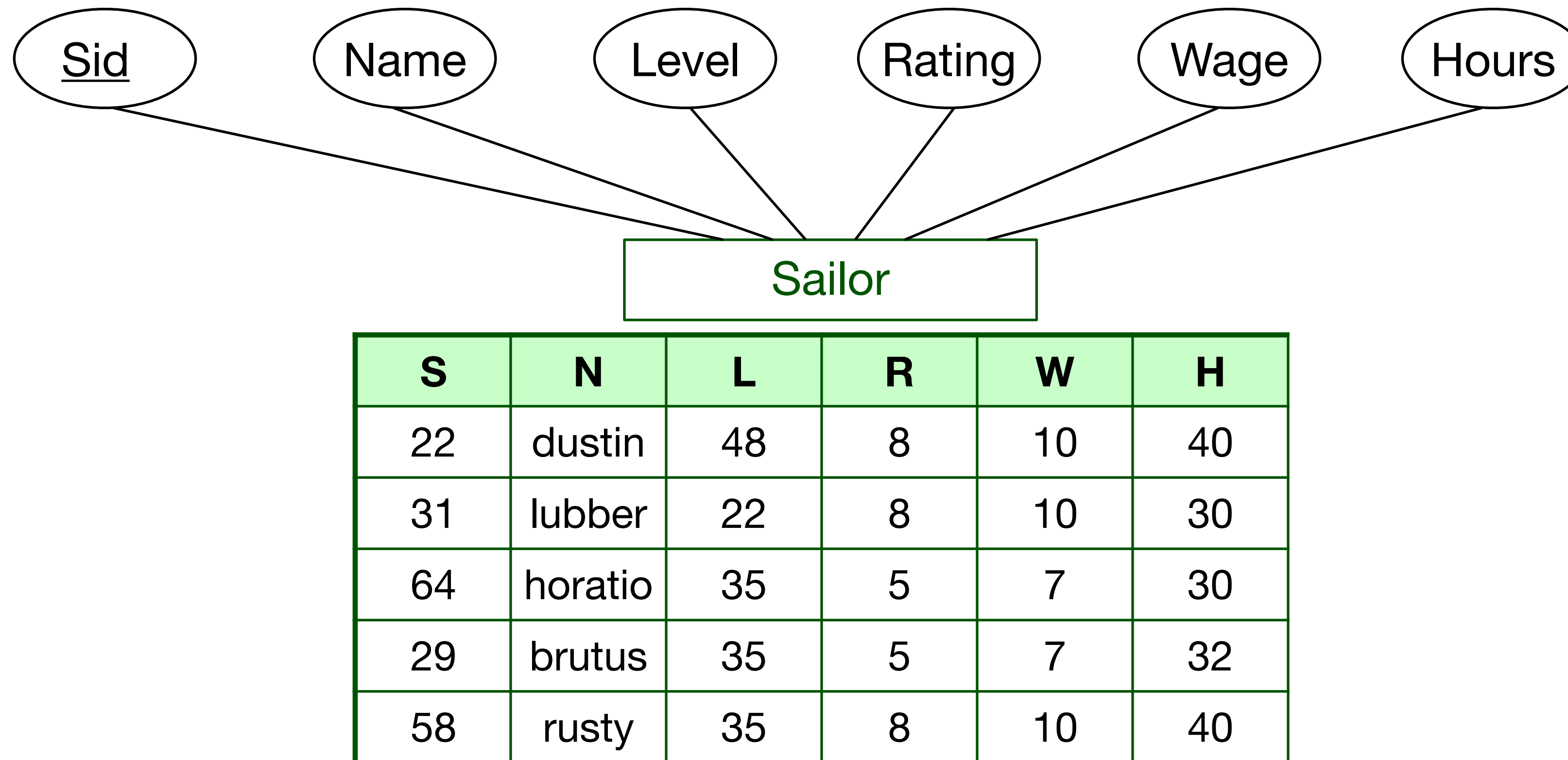
```
CREATE TABLE Reserves (  
    sname CHAR(10),  
    bid INTEGER,  
    day DATE,  
    PRIMARY KEY (bid,day),  
    CONSTRAINT noInterlakeRes  
    CHECK ('Interlake' <>  
        (SELECT B.bname  
         FROM Boats B  
         WHERE B.bid=bid)))
```

# Constraints Over Multiple Relations

Number of boats plus number of sailors is  $< 100$

```
CREATE ASSERTION smallClub  
CHECK  
( (SELECT COUNT (S.sid) FROM Sailors S)  
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

# Data Redundancy



- **Constraint:** all sailors with the same rating have the same wage ( $R \rightarrow W$ )
- Problems due to data redundancy?

# Problems due to Data Redundancy

- Problems due to  $R \rightarrow W$  :
  - Update anomaly: Can we change W in just the first tuple of SNLRWH?
  - Insertion anomaly: What if we want to insert an employee and don't know the hourly wage for his rating?
  - Deletion anomaly: If we delete all employees with rating 5, we lose the information about the wage for rating 5!
- Solution?

# Relation Decomposition

S	N	L	R	W	H
22	dustin	48	8	10	40
31	lubber	22	8	10	30
64	horatio	35	5	7	30
29	brutus	35	5	7	32
58	rusty	35	8	10	40

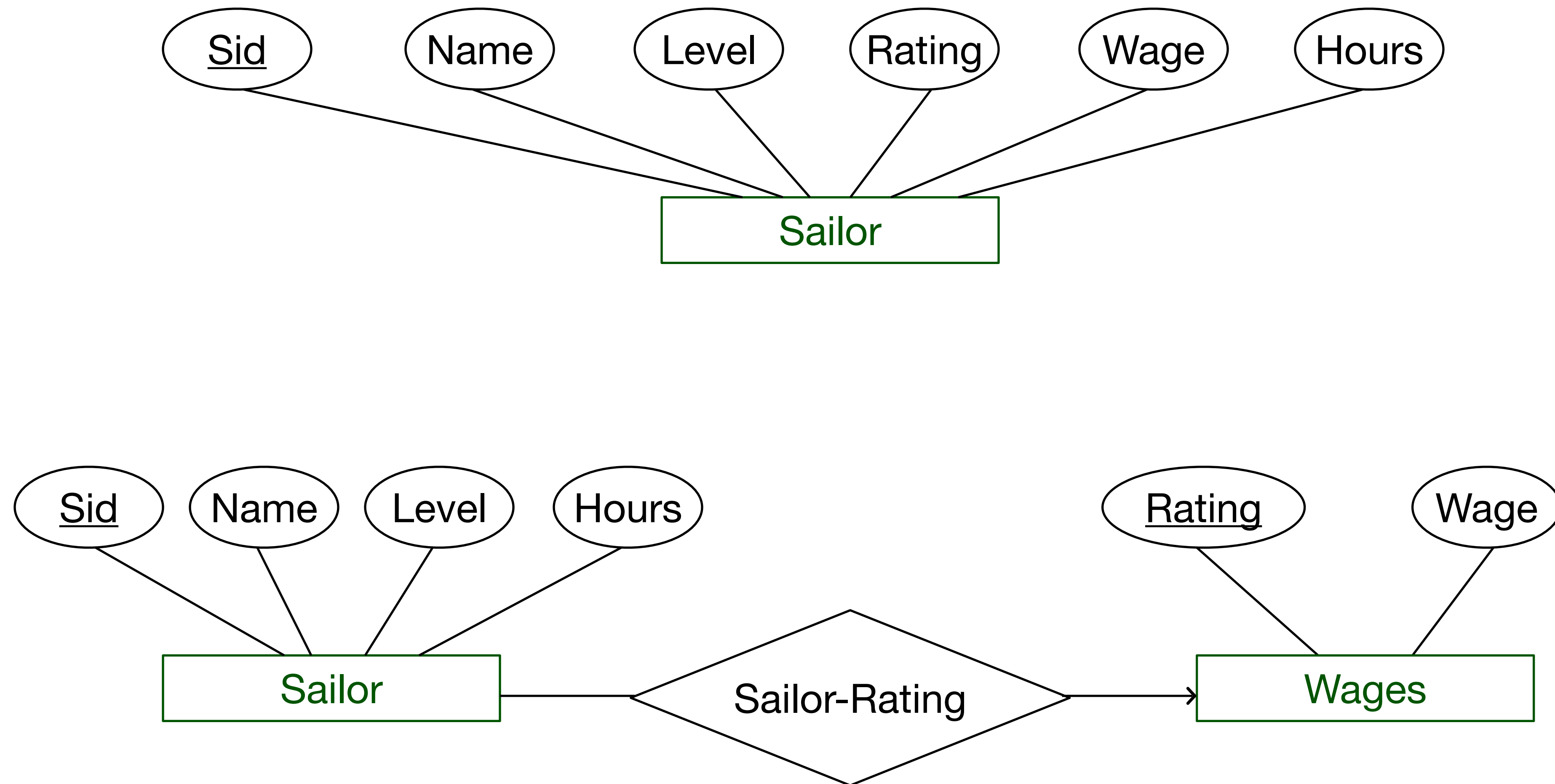
S	N	L	R	H
22	dustin	48	8	40
31	lubber	22	8	30
64	horatio	35	5	30
29	brutus	35	5	32
58	rusty	35	8	40

Wages

R	W
8	10
5	7

Problem?

# Recall: Modifying ER Diagram



# Towards Normal Forms


- First question is to ask whether any schema refinement is needed
- If a relation is in a **normal form** (BCNF, 3NF etc.), certain anomalies are avoided/minimized
- If not, decompose relation to normal form
- Role of **functional dependencies** (FDs) in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC
    - **No FDs hold:** There is no redundancy here.
    - **Given  $A \rightarrow B$ :** Several tuples could have the same A value, and if so, they'll all have the same B value!



# Functional Dependencies (FDs)

- A functional dependency  $X \rightarrow Y$  between sets of attributes  $X$  and  $Y$  holds over relation  $R$  if, for every allowable instance  $r$  of  $R$ :
  - $\pi_X(t) = \pi_X(u)$  implies  $\pi_Y(t) = \pi_Y(u)$  for all  $t \in r$  and  $u \in r$
  - i.e., given two tuples in  $r$ , if the  $X$  values agree, then the  $Y$  values must also agree
- An FD is a statement about **all** allowable instances.
  - Must be identified based on the application's semantics
  - Given allowable instance  $r$  of  $R$ , we can check if  $r$  violates some FD  $f$ , but we cannot tell if  $f$  holds over  $R$ !
- $K$  is a candidate key for  $R$  means that  $K \rightarrow R$ 
  - However,  $K \rightarrow R$  does not require  $K$  to be minimal!

# Reasoning About FDs

- Given some FDs, we can usually infer additional FDs:
  - $ssn \rightarrow did, did \rightarrow lot$  implies  $ssn \rightarrow lot$
- An FD  $f$  is implied by a set of FDs  $F$  if  $f$  holds whenever all FDs in  $F$  hold.
  - $F^+ =$  closure of  $F$  is the set of all FDs that are implied by  $F$ .
- Armstrong's Axioms ( $X, Y, Z$  are sets of attributes):
  - Reflexivity: If  $Y \subseteq X$ , then  $X \rightarrow Y$   trivial FDs
  - Augmentation: If  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$  for any  $Z$
  - Transitivity: If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$
- These are sound and complete inference rules for FDs!

# Reasoning About FDs (Contd.)

- Couple of additional rules (that follow from Armstrongs' axioms):
  - **Union:** If  $X \rightarrow Y$  and  $X \rightarrow Z$ , then  $X \rightarrow YZ$
  - **Decomposition:** If  $X \rightarrow YZ$ , then  $X \rightarrow Y$  and  $X \rightarrow Z$
- Example: **Contracts(cid,sid,jid,did,pid,qty,value)**, and:
  - C is the key:  **$C \rightarrow CSJDPQV$**
  - Project purchases each part using single contract:  **$JP \rightarrow C$**
  - Department purchases at most one part from a supplier:  **$SD \rightarrow P$**
- **Can you infer  $SDJ \rightarrow CSJDPQV$  ?**

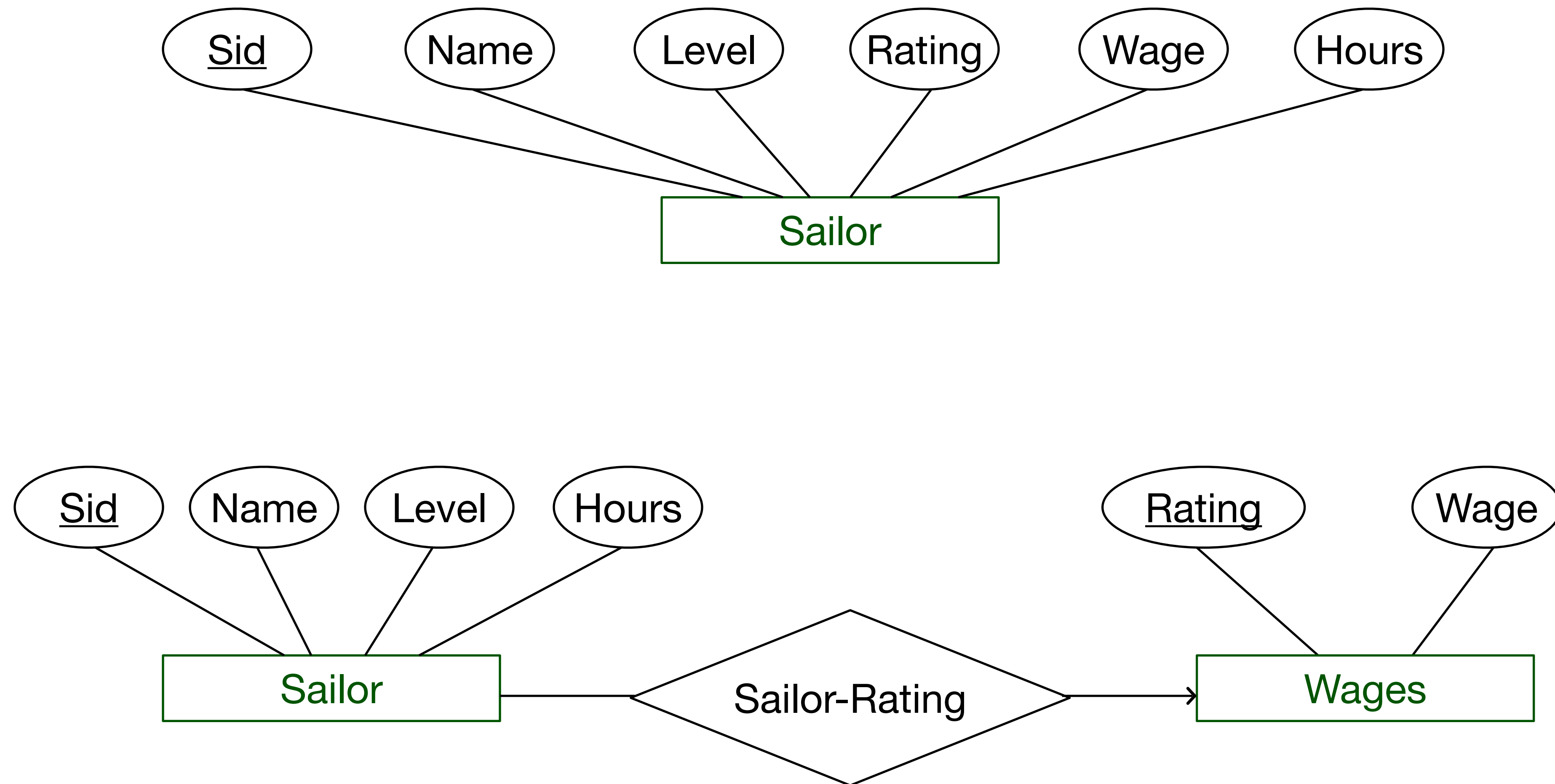
# Reasoning About FDs (Contd.)

- Computing the closure of a set of FDs can be expensive.  
(Size of closure is exponential in # attrs!)
- Typically, we just want to check if a given FD  $X \rightarrow Y$  is in the closure of a set of FDs  $F$ . An efficient check:
  - Compute attribute closure of  $X$  (denoted  $X^+$ ) wrt  $F$ :
    - Set of all attributes  $A$  such that  $X \rightarrow A$  is in  $F^+$
    - There is a linear time algorithm to compute this.
  - Check if  $Y$  is in  $X^+$
- Does  $F = \{A \rightarrow B, B \rightarrow C, CD \rightarrow E\}$  imply  $A \rightarrow E$ ?
  - i.e., is  $A \rightarrow E$  in the closure  $F^+$ ? Equivalently, is  $E$  in  $A^+$ ?
  - Can be used to find keys!!!

# Finding Keys

- Example: **Contracts(cid,sid,jid,did,pid,qty,value)**, and:
  - C is the key:  **$C \rightarrow CSJDPQV$**
  - Project purchases each part using single contract:  **$JP \rightarrow C$**
  - Department purchases at most one part from a supplier:  **$SD \rightarrow P$**
- Discussion: Find all the keys of Contracts!
  - How do you go about it?

# Recall: Modifying ER Diagram



# Decomposition of a Relation Scheme

- Suppose that relation R contains attributes A1 ... An. A decomposition of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
  - Every attribute of R appears as an attribute of one of the new relations.
- Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- E.g., can decompose **SNLRWH** into **SNLRH** and **RW**.

# Example Decomposition

- Decompositions should be used only when needed.
  - SNLRWH has FDs  $S \rightarrow \text{SNLRWH}$  and  $R \rightarrow W$
  - Data duplication due to second FD
  - Will make this more precise during the definition of normal forms
- Decompose to SNLRH and RW
  - What should we be careful about?



# Problems with Decompositions

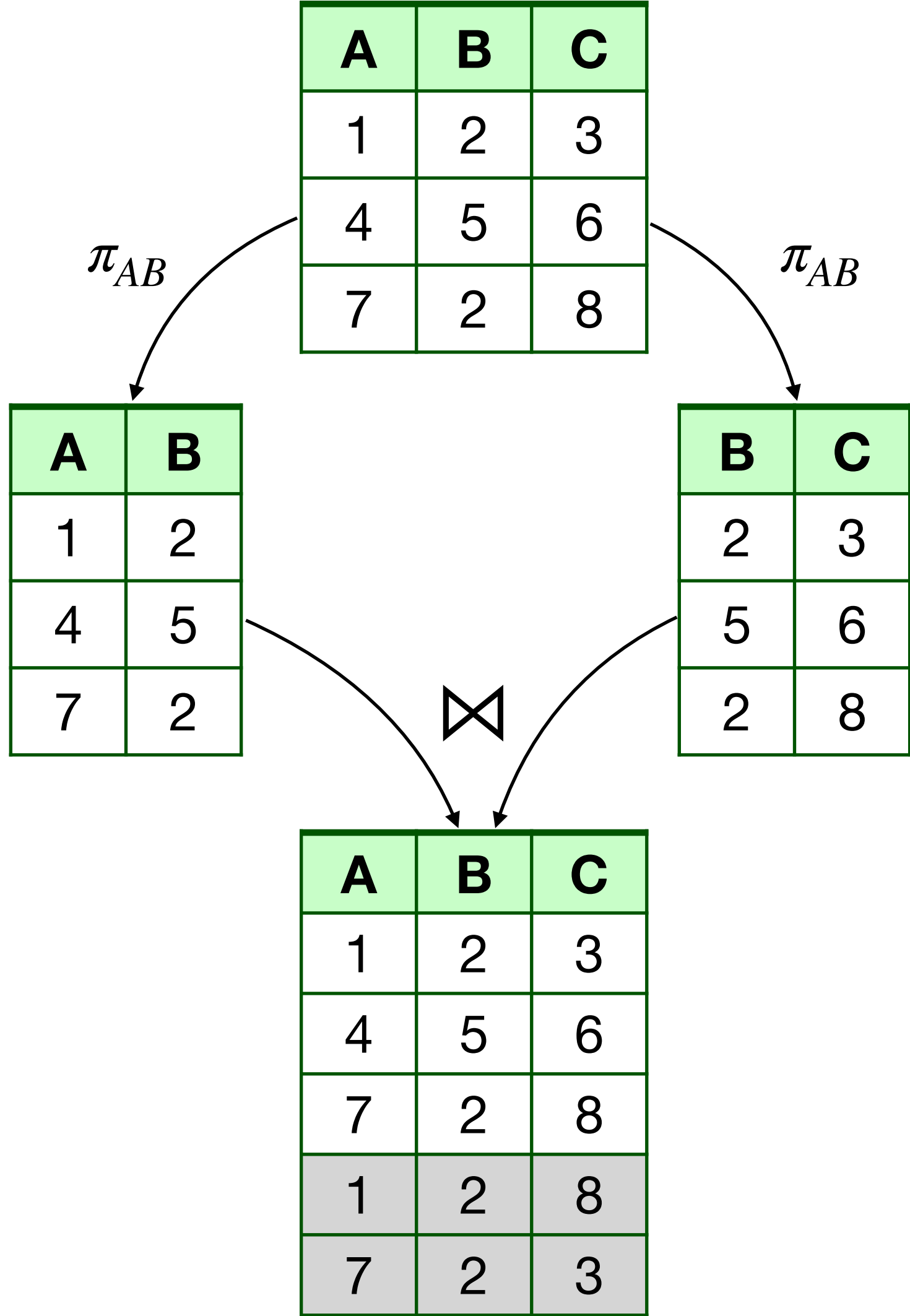
- There are three potential problems to consider:
  1. Some queries become more **expensive**
    - e.g., How much did sailor Joe earn? (salary =  $W \cdot H$ )
  2. Given instances of the decomposed relations, we may **not be able to reconstruct** the corresponding instance of the original relation
    - Fortunately, not in the SNLRWH example.
  3. **Checking dependencies may require joining** the instances of the decomposed relations
    - Fortunately, not in the SNLRWH example.
- Tradeoff: Must consider these issues vs. redundancy.

# Lossless-Join Decompositions

- Decomposition of  $R$  into  $X$  and  $Y$  is **lossless-join** w.r.t. a set of FDs  $F$  if, for every instance  $r$  that satisfies  $F$ :  $\pi_X(r) \bowtie \pi_Y(r) = r$
- It is always true that  $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$ 
  - In general, the other direction does not hold!
- Definition extended to decomposition into 3 or more relations in a straightforward way.
- It is essential that all decompositions used to deal with redundancy be lossless-join! (Avoids Problem 2.)

# More on Lossless-Join

- The decomposition of R into X and Y is **lossless-join w.r.t. F if and only if** the  $F^+$  contains:
  - $X \cap Y \rightarrow X$ , or
  - $X \cap Y \rightarrow Y$
- In particular, the decomposition of R into UV and R - V is lossless-join if  $U \rightarrow V$  holds over R

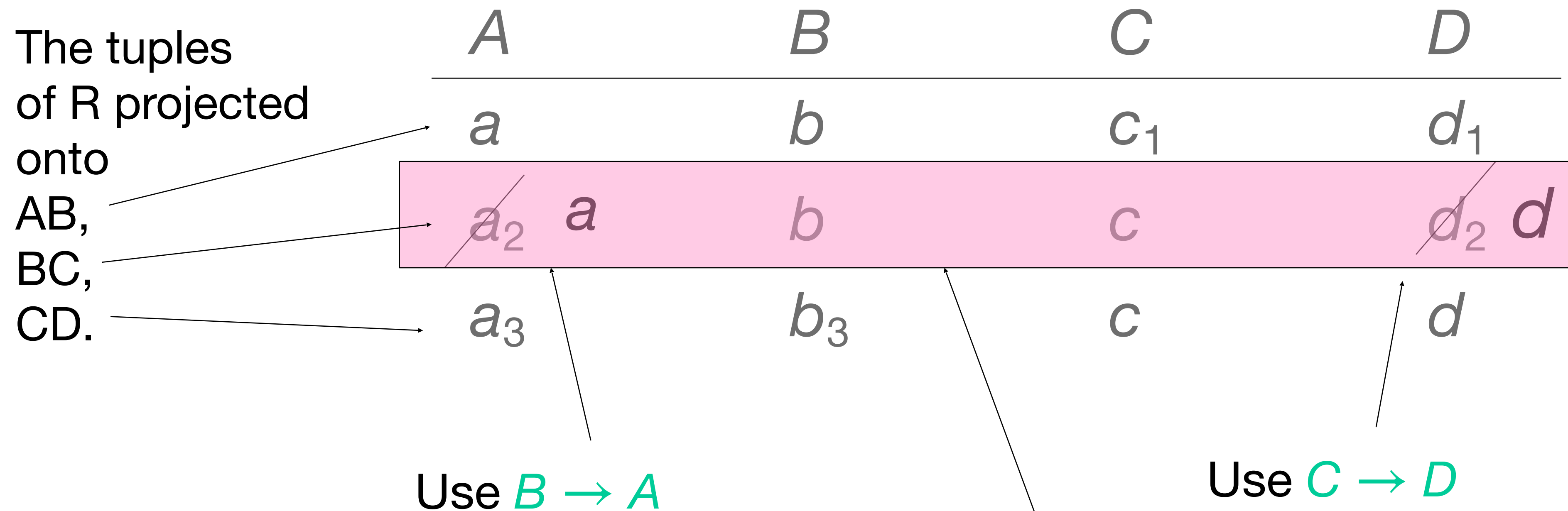


# The Chase Test

- Suppose tuple  $t$  comes back in the join.
  - Start by assuming  $t = abc\dots$
- Then  $t$  is the join of projections of some tuples of  $R$ , one for each  $R_i$  of the decomposition.
  - For each  $i$ , there is a tuple  $s_i$  of  $R$  that has  $a, b, c, \dots$  in the attributes of  $R_i$ .
  - $s_i$  can have any values in other attributes.
- Can we use the given FD's to show that one of these tuples must be  $t$ ?

# Example: The Chase

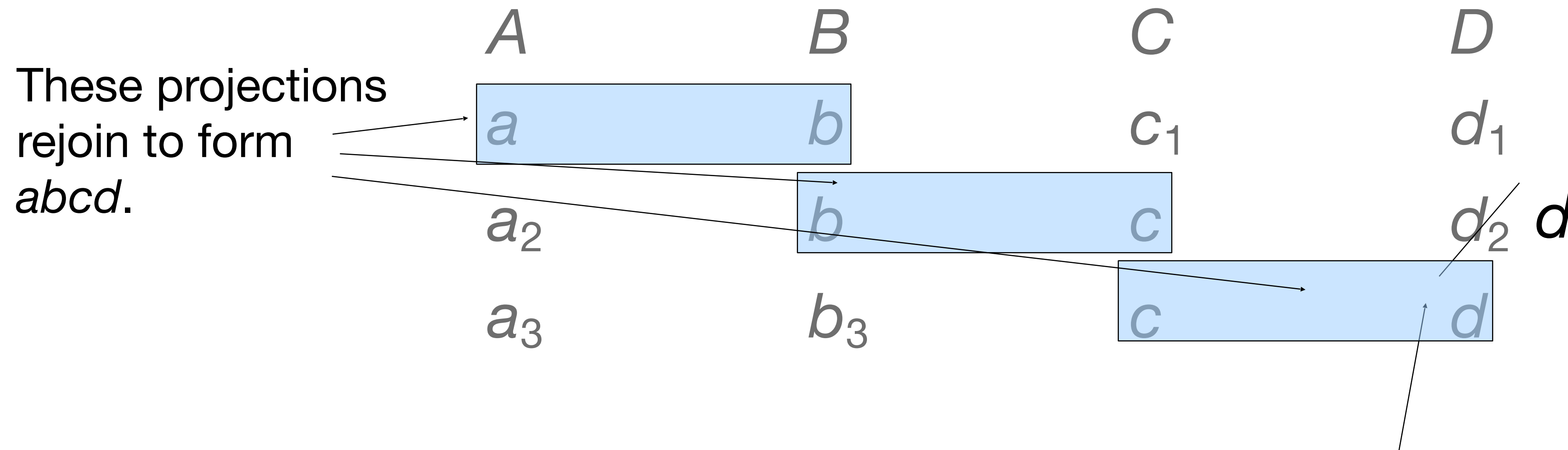
- Let  $R = ABCD$ , and the decomposition be  $AB$ ,  $BC$ , and  $CD$ .
- Let the given FD's be  $C \rightarrow D$  and  $B \rightarrow A$ .
- Suppose the tuple  $\mathbf{t} = \mathbf{abcd}$  is the join of tuples projected onto  $AB$ ,  $BC$ ,  $CD$ .



We've proved the second tuple must be  $t$ .

# Example: Lossy Join

- Same relation  $R = ABCD$  and same decomposition.
- But with only the FD  $C \rightarrow D$ .



These three tuples are an example that shows the join is actually lossy. Tuple  $abcd$  is not in  $R$ , but we can project and rejoin to get  $abcd$ .

Use  $C \rightarrow D$

# Dependency-Preserving Decomposition

- Consider CSJDPQV, C is key,  $JP \rightarrow C$  and  $SD \rightarrow P$ .
  - Decomposition: CSJDQV and SDP
  - (Is it lossless-join?)
  - Problem: Checking  $JP \rightarrow C$  requires a join!
- **Dependency-preserving decomposition** (Intuitive):
  - If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold. (Avoids Problem (3).)
- Projection of set of FDs F: Projection of F onto X (denoted  $F_X$ ) is the set of FDs  $U \rightarrow V \in F^+$  (F's closure) such that  $U, V \subseteq X$ .



# Dependency Preserving Decompositions (Contd.)

- Decomposition of R into X and Y is dependency preserving if  $(F_X \cup F_Y)^+ = F^+$ 
  - i.e., if we consider only dependencies in the closure  $F^+$  that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in  $F^+$ .
- Important to consider  $F^+$ , **not**  $F$ , in this definition:
  - ABC,  $A \rightarrow B$ ,  $B \rightarrow C$ ,  $C \rightarrow A$ , decomposed into AB and BC.
  - Is this dependency preserving? Is  $C \rightarrow A$  preserved?
- Dependency preserving does not imply lossless-join:
  - ABC,  $A \rightarrow B$ , decomposed into AB and BC. (see first lossy join example)
- And vice-versa!

# What should we learn today?

- Explain the semantics of **aggregation** operations and **grouping** in SQL
- Formulate queries that combine **joins**, **sub-queries**, **grouping**, and **aggregation**
- Define and explain the notion of **functional dependencies** (FDs), and apply rules to reason about FDs including Armstrong's Axioms
- Review the notion of **keys** and how to **determine** them by reasoning on functional dependencies
- Explain the issues in **decomposition** of relations and verify whether decompositions are **lossless-join** and **dependency-preserving**
- Apply the **chase test** to determine if a decomposition is lossless-join

