Databases and Information Systems

Extended Relational Algebra
SQL

Dmitriy Traytel
slides partly by Marcos Vaz Salles

UNIVERSITY OF COPENHAGEN

Do-It-Yourself Recap: Selection and Projection

- What were the semantics of the selection and projection operators? Can they be composed? Why?

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Source: Ramakrishnan & Gehrke

DIKU

Do-It-Yourself Recap: Selection and Projection

- What were the semantics of the selection and projection operators? Can they be composed? Why?

$\sigma_{rating>8}(S2)$

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Source: Ramakrishnan & Gehrke

Do-It-Yourself Recap: Selection and Projection

- What were the semantics of the selection and projection operators? Can they be composed? Why?

$$\sigma_{rating>8}(S2)$$

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

Relational Algebra

- Basic operations:
  - Selection σ Selects a subset of rows from relation
  - Projection π Deletes unwanted columns from relation
  - Cross-product $\times$ Allows us to combine two relations
  - Set-difference $-$ Tuples in relation 1, but not in relation 2
  - Union $\cup$ Tuples in relation 1 and in relation 2
- Additional operations:
  - Intersection $\cap$, join $\bowtie$, antijoin $\triangleright$, division $\div$, renaming ρ: Not essential, but (very!) useful.
- Each operation returns a finite relation, i.e., operations can be composed! (Algebra is “closed”.)

Relational Query Languages

- Query Languages != programming languages!
  - QLs not expected to be "Turing complete".
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.
- Relational Algebra has limited expressibility
  - What queries are hard to answer?
  - What queries can't we answer?

Source: Ramakrishnan & Gehrke (partial)

Hard, but possible

- Relational Schema:
  - Employees(ssn: integer; sal: real; mgr_ssn: integer).
- **Find the employees with the highest salary:**
  - How would you do it?
- Intuition:
  - Create “dominance” relation: e1 dominates e2 if e1 has salary higher than or equal to e2
  - Divide “dominance” relation by original table: Finds employees who dominate all others
- What about the second highest salary?

Impossible, but very useful

- Relational Schema:
  - Employees(ssn: integer; sal: real; mgr_ssn: integer).
  - Works_In(ssn: integer; did: integer)
- Find the total amount paid in salaries by department
- Problem: **We do not know how to count!**
- What about finding employees that work in exactly three departments?

Impossible, but very useful

- Relational Schema:
  - Employees(<u>ssn</u>: integer; sal: real; mgr_ssn: integer).
  - Works_In(<u>ssn</u>: integer; <u>did</u>: integer)
- Find all the superiors of employee with SSN 123-22-3666
- We want a **Transitive Closure**: Comes in handy when you query graphs
- **Problem: We can't write arbitrary loops/recursion!**
  - RA operators only implement implicit "foreach element in relation" loops

Extensions to Relational Algebra

- Relational query languages restrict expressiveness to obtain ease of use and of optimization.

- There are some very useful queries we cannot express in the relational algebra.

- Many extensions proposed to handle those queries made into SQL.

We will study an extended relational algebra next!

Extensions to Relational Algebra

- Relational query languages restrict expressiveness to obtain ease of use and of optimization.

- There are some very useful queries we cannot express in the relational algebra.

- Many extensions proposed to handle those queries made into SQL.

We will study an extended relational algebra next!

- Expressions in projections
- Bags vs sets
- Duplicate elimination
- Grouping/aggregation
- Sorting
- Outer joins
- ~~Recursion~~

Source: Ramakrishnan & Gehrke (partial)

Extending Projection with Expressions

- Using the same π_L operator, we allow the list L to contain arbitrary expressions involving attributes:
  - Arithmetic on attributes, e.g., A+B→C.
  - Duplicate occurrences of the same attribute.

$\pi_{\text{rating+sid} \rightarrow \text{rs,age,age}}(S1)$

rs	age	age
29	45.0	45.0
39	55.5	55.5
68	35.0	35.0

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Source: Bulskov (partial)

Relational Algebra on Bags

- A bag (or multiset) is an unordered collection where duplicates are allowed
  - Like a set, but an element may appear more than once.
- Example: {1,2,1,3} is a bag.
- Example: {1,2,3} is a bag that happens to also be a set.
- SQL uses bag semantics

$\pi_{age}(S2)$

age
35.0
55.5
35.0
35.0

$\sigma_{age<40.0}(\pi_{age}(S2))$

age
35.0
35.0
35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Source: Bulskov (partial)

Bag Product

$\sigma_{age<40.0}(\pi_{age}(S2))$

age
35.0
35.0
35.0

$\pi_{sname}(\sigma_{sid<40}(S2))$

sname
yuppy
lubber

$\sigma_{age<40.0}(\pi_{age}(S2)) \times \pi_{sname}(\sigma_{sid<40}(S2))$

age	sname
35.0	yuppy
35.0	yuppy
35.0	yuppy
35.0	lubber
35.0	lubber
35.0	lubber

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Source: Bulskov

Bag Union, Intersection, and Difference

- An element appears in the union of two bags the sum of the number of times it appears in each bag.
  - Example: {1,2,1} ∪ {1,1,2,3,1} = {1,1,1,1,1,2,2,3}

- An element appears in the intersection of two bags the minimum of the number of times it appears in either.
  - Example: {1,2,1,1} ∩ {1,2,1,3} = {1,1,2}.

- An element appears in the difference A — B of bags as many times as it appears in A, minus the number of times it appears in B, but never less than 0 times.
  - Example: {1,2,1,1} — {1,2,3} = {1,1}.

3 − 1 = 2, 1 − 1 = 0 (2)

Beware: Bag Laws != Set Laws

- Some, but not all algebraic laws that hold for sets also hold for bags.
- Examples
  - The commutative law for union R ∪ S = S ∪ R holds for bags, since addition is commutative
  - However, set union is idempotent, meaning that S ∪ S = S.
  - For bags, if x appears n times in S, then it appears 2n times in S ∪ S.
  - Thus S ∪ S != S in general, e.g., {1} ∪ {1} = {1,1} != {1}.

Source: Bulskov (partial)

Duplicate Elimination

- R1 := δ(R2).
- R1 consists of one copy of each tuple that appears in R2 one or more times.

π_{age}(S2)

age
35.0
55.5
35.0
35.0

$\delta(\pi_{age}$(S2))

age
35.0
55.5

Source: Bulskov (partial)

Aggregation Operators

- Aggregation operators are not operators of relational algebra.
- Rather, they apply to entire columns of a table and produce a single result.
- The most important examples: SUM, AVG, COUNT, MIN, and MAX.

SUM(rating) = 32
COUNT(sid) = 4
MAX(age) = 55.5
AVG(rating) = 8

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Source: Bulskov (partial)

Grouping Operator

- $\gamma_L(R)$ where L is a list of elements that are either:
  - Individual (grouping) attributes.
  - AGG(A), where AGG is one of the aggregation operators and A is an attribute.
    - An arrow and a new attribute name renames the component.

- Semantics
  - Form one group for each distinct list of values in R for grouping attributes in list L.
  - Within each group, compute AGG(A) for each aggregation on list L.
  - Result has one tuple for each group:
    - The grouping attributes and
    - Their group's aggregations.

$\gamma_{\text{age,COUNT(rating)}\rightarrow \text{cr}}(S2)$

cr	age
3	35.0
1	55.5

$\gamma_{\text{age,MAX(rating)}\rightarrow \text{mr}}(S2)$

mr	age
10	35.0
8	55.5

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Source: Bulskov (partial)

Outer Join

- Suppose we join R $\bowtie$ S.
- A tuple of R that has no tuple of S with which it joins is called dangling.
  - Similarly for a tuple of S.
- Outer join $\overset{\circ}{\bowtie}$ preserves dangling tuples by padding them with $\perp$ (NULL in SQL).
- Variants that preserve only left/right dangling tuples: $\overset{\circ}{\bowtie}_L$, $\overset{\circ}{\bowtie}_R$

$$S1 \overset{\circ}{\bowtie}_L R1 = S1 \overset{\circ}{\bowtie} R1$$

$$S1 \overset{\circ}{\bowtie}_R R1 = S1 \bowtie R1$$

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10.10
58	rusty	10	35.0	103	11.12
31	lubber	8	55.5	$\perp$	$\perp$

R1

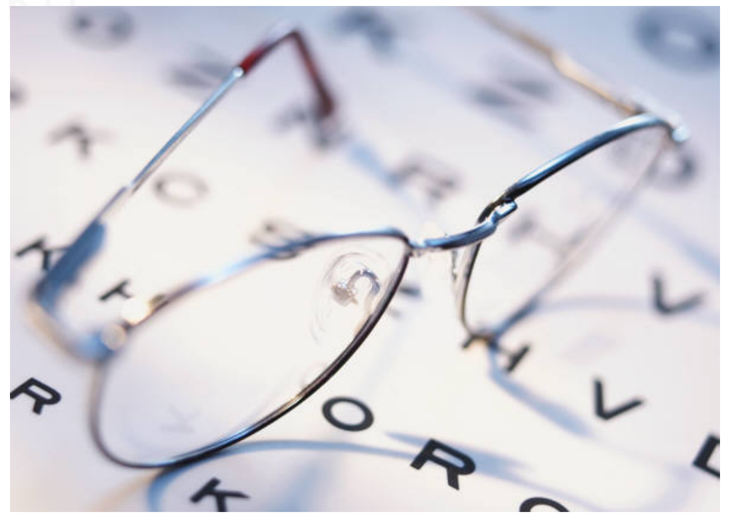
sid	bid	day
22	101	10.10
58	103	11.12

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Source: Bulskov (partial)

What should we learn today?



- Formulate basic SQL queries, namely **select-project-join** queries
- Explain issues related to **duplicates** in SQL
- Formulate SQL statements to **insert**, **delete**, and **update** data
- Explain the semantics of the varied types of **joins** that can be formulated in SQL
- Explain the semantics of **NULL** in SQL and reason about expressions in a three-valued logic
- Identify the multiple constructions for **sub-queries** in SQL and explain their meaning
- Explain the semantics of **aggregation** operations and **grouping** in SQL
- Formulate queries that combine **joins**, **sub-queries**, **grouping**, & **aggregation**

Query Languages

- Relational Calculus ➡ **declarative**: describe the result, not how to compute it
- Relational Algebra ➡ **operational**: describe the computation as sequence of algebraic transformations
- SQL ➡ start with the above and move to the "**real**" world

Query Languages

- Relational Calculus ➡ **declarative**: describe the result, not how to compute it
- Relational Algebra ➡ **operational**: describe the computation as sequence of algebraic transformations
- SQL ➡ start with the above and move to the “**real**” world

Basic SQL Query

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

```
SELECT   S.Name
FROM     Sailors S
WHERE    S.Age > 25
```

```
SELECT DISTINCT S.Name
FROM    Sailors S
WHERE   S.Age > 25
```

- Default is that duplicates are not eliminated!
  - Need to explicitly say “DISTINCT”

Select-Project-Join (SPJ) Queries

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

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Reserves

sid	bid	day
22	101	10.10
58	103	11.12

Source: Ramakrishnan & Gehrke (partial)

Conceptual Evaluation

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

sid	sname	rating	age	sid	bid	day
22	dustin	7	45.0	22	101	10.10
22	dustin	7	45.0	58	103	11.12
31	lubber	8	55.5	22	101	10.10
31	lubber	8	55.5	58	103	11.12
58	rusty	10	35.0	22	101	10.10
58	rusty	10	35.0	58	103	11.12

Source: Ramakrishnan & Gehrke

Conceptual Evaluation

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

sid	sname	rating	age	sid	bid	day
22	dustin	7	45.0	22	101	10.10
22	dustin	7	45.0	58	103	11.12
31	lubber	8	55.5	22	101	10.10
31	lubber	8	55.5	58	103	11.12
58	rusty	10	35.0	22	101	10.10
58	rusty	10	35.0	58	103	11.12

Source: Ramakrishnan & Gehrke

Conceptual Evaluation

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

sid	sname	rating	age	sid	bid	day
22	dustin	7	45.0	22	101	10.10
22	dustin	7	45.0	58	103	11.12
31	lubber	8	55.5	22	101	10.10
31	lubber	8	55.5	58	103	11.12
58	rusty	10	35.0	22	101	10.10
58	rusty	10	35.0	58	103	11.12

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.

Source: Ramakrishnan & Gehrke

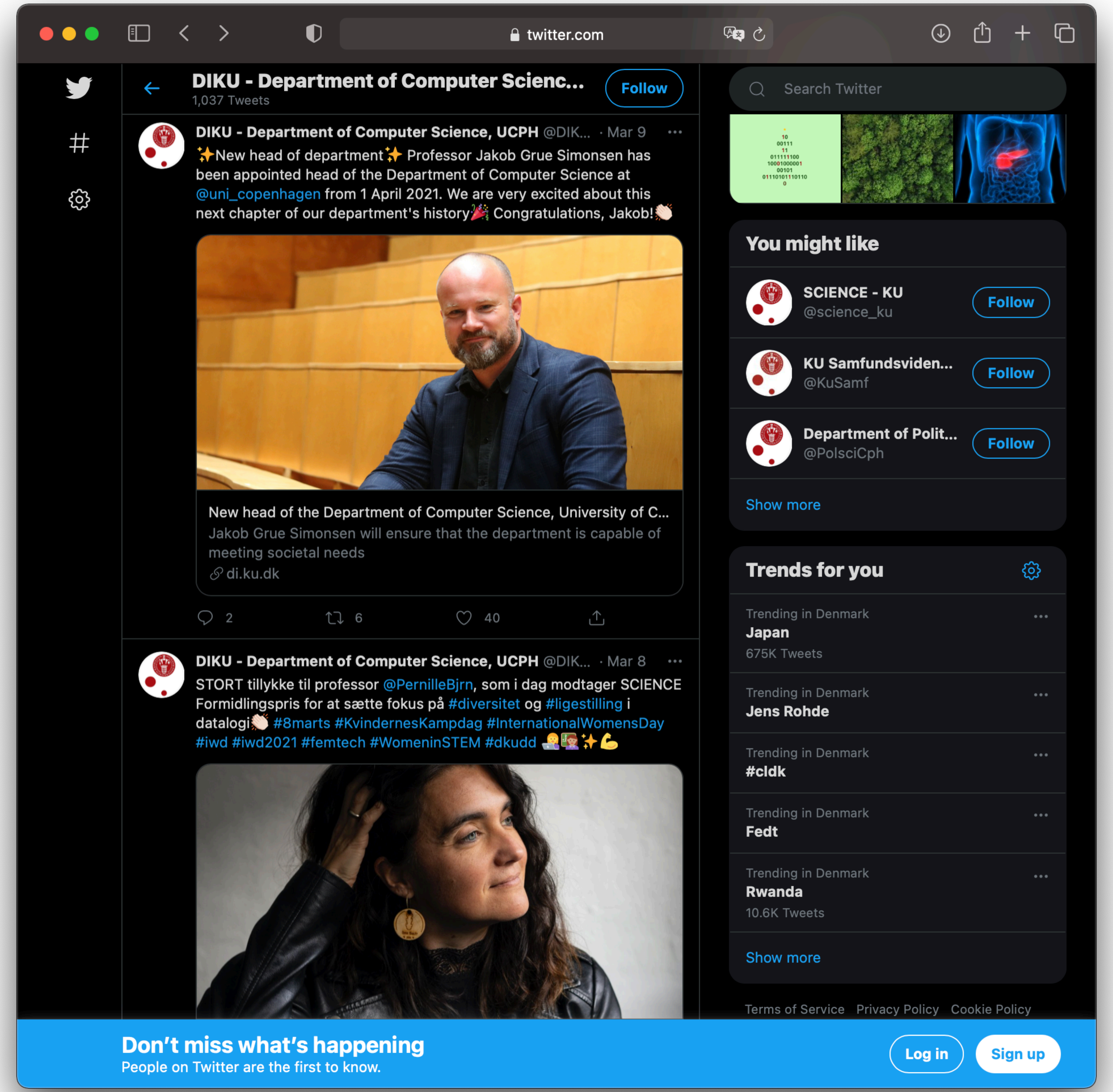
Algorithmic View of Conceptual Evaluation

```
foreach t1 in R1 {
  foreach t2 in R2 {
  …
   foreach tn in Rn {
     cand := t1 , t2 , … , tn;
     if (CONDITION(cand)) {
       cand := PROJECT(cand);
       result := DISTINCT ? result ∪ cand : result ∪all cand;
     }
   }
  …
  }
}
```

Queries in SQL

- Let's say we model Twitter using the following three tables:
  - Users(uid, name, joineddate)
  - Posts(pid, uid, date, text)
  - Mentions(pid, uid)

- Write the following queries in SQL:
  - List the names of users who joined after 1/1/2020, who posted anything before 1/1/2022
  - List the texts of all posts of users named "Dmitriy" that mentioned users named "MC"



A Slightly Modified Query

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

- Would adding DISTINCT to this query make a difference?

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

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

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

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

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

Source: Ramakrishnan & Gehrke

How are these different?

- Find sid's of sailors who've reserved a red or a green boat

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

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

- Query on the right does not perform duplicate elimination

Source: Ramakrishnan & Gehrke

Find sid's of sailors who've reserved both 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'
```

- What if INTERSECT were replaced by EXCEPT?
  - EXCEPT is set difference

A different way to say it

Both and (intersect)

```
SELECT   DISTINCT S.sid
FROM     Sailors S, Boats B1, Reserves R1,
                    Boats B2, Reserves R2
WHERE    S.sid=R1.sid AND R1.bid=B1.bid
  AND    S.sid=R2.sid AND R2.bid=B2.bid
  AND    B1.color='red' AND B2.color='green'
```

Adding and Deleting Tuples

```
INSERT INTO Table
  VALUES(A1,A2,…,An)
```

```
INSERT INTO Table
  Select-Statement
```

- Can insert a single tuple using:

```
INSERT INTO  Students (sid, name, login, age, gpa)
VALUES  (53688, ‘Smith’, ‘smith@ee’, 18, 3.2)
```

- Can insert a set of tuples using:

```
INSERT INTO Students(sid, name, login, age, gpa)
SELECT NULL, name, login, age, 0.0
FROM   Other_Students
WHERE  school = ‘KU’
```

Source: Widom (partial) / Ramakrishnan & Gehrke (partial)

Adding and Deleting Tuples

```
INSERT INTO Table
  VALUES(A1,A2,…,An)
```

```
INSERT INTO Table
  Select-Statement
```

- Can insert a single tuple using:

```
INSERT INTO  Students (sid, name, login, age, gpa)
VALUES  (53688, ‘Smith’, ‘smith@ee’, 18, 3.2)
```

- Can insert a set of tuples using:

assumes IDENTITY column

```
INSERT INTO Students(sid, name, login, age, gpa)
SELECT NULL, name, login, age, 0.0
FROM   Other_Students
WHERE  school = ‘KU’
```

Adding and Deleting Tuples

```
DELETE FROM Table
WHERE Condition
```

- Can delete all tuples satisfying some condition (e.g., name = Smith):

```
DELETE
FROM Students S
WHERE S.name = ‘Smith’
```

Updating Tuples

UPDATE *Table*
SET $A_1 = Expr_1, A_2 = Expr_2, \ldots, A_n = Expr_n$
WHERE *Condition*

- Can update all tuples satisfying some condition (e.g., age >= 36):

```
UPDATE Employees
SET salary = salary * 1.1
WHERE age >= 36
```

Cross Join = Cartesian Product

MovieStar	
name	**address**
Harrison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Mads Mikkelsen	Sverresgata 23

MovieExec	
name	**address**
Harison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Sandra Bullock	564 Center B

```
SELECT S.*, E.*
FROM MovieStar S, MovieExec E
```

- If no WHERE clause is specified evaluates to the N-ary bag product S × E

Cross Join = Cartesian Product

MovieStar	
name	**address**
Harrison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Mads Mikkelsen	Sverresgata 23

```
SELECT *
FROM MovieStar CROSS JOIN MovieExec
```

MovieExec	
name	**address**
Harison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Sandra Bullock	564 Center B

name	address	name	address
Harrison Ford	789 Palm	Harison Ford	789 Palm
Harrison Ford	789 Palm	Iben Hjejle	Øster Alle 4
Harrison Ford	789 Palm	Sandra Bullock	564 Center B
Iben Hjejle	Øster Alle 4	Harison Ford	789 Palm
Iben Hjejle	Øster Alle 4	Iben Hjejle	Øster Alle 4
Iben Hjejle	Øster Alle 4	Sandra Bullock	564 Center B
Mads Mikkelsen	Sverresgata 23	Harison Ford	789 Palm
Mads Mikkelsen	Sverresgata 23	Iben Hjejle	Øster Alle 4
Mads Mikkelsen	Sverresgata 23	Sandra Bullock	564 Center B

Source: Bulskov & Lassen (partial)

Joins in SQL

MovieStar	
name	**address**
Harrison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Mads Mikkelsen	Sverresgata 23

MovieExec	
name	**address**
Harison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Sandra Bullock	564 Center B

```
SELECT S.*, E.*
FROM   MovieStar S,
       MovieExec E
WHERE  S.name = E.name
```

```
SELECT S.*, E.*
FROM   MovieStar S
       JOIN MovieExec E
       ON S.name = E.name
```

- The join of two relations (R1,R2)
- Matching tuples are returned
- Corresponds to a theta join

S.name	S.address	E.name	E.address
Harrison Ford	789 Palm	Harrison Ford	789 Palm
Iben Hjejle	Øster Alle 4	Iben Hjejle	Øster Alle 4

Source: Bulskov & Lassen (partial)

Alternative Syntax for Equijoins

MovieStar	
name	**address**
Harrison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Mads Mikkelsen	Sverresgata 23

```
SELECT *
FROM   MovieStar S
       INNER JOIN MovieExec E
       USING (name)
```

MovieExec	
name	**address**
Harison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Sandra Bullock	564 Center B

- Equijoin of two relations (R1,R2) specified with INNER JOIN / USING
- Matching tuples are returned, but equijoin columns coalesced

single

name	address	address
Harrison Ford	789 Palm	789 Palm
Iben Hjejle	Øster Alle 4	Øster Alle 4

Natural Joins

MovieStar	
name	**address**
Harrison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Mads Mikkelsen	Sverresgata 23

```
SELECT *
FROM MovieStar
     NATURAL JOIN MovieExec
```

MovieExec	
name	**address**
Harison Ford	789 Palm
Iben Hjejle	Øster Alle 4
Sandra Bullock	564 Center B

- In the **natural join** the USING clause is implicit
- matching attribute-names are taken as join columns

name	address
Harrison Ford	789 Palm
Iben Hjejle	Øster Alle 4

```
SELECT b.boat,
       r.regatta_name
FROM   boats b,
       entries e,
       races r
WHERE boat = 'svuppe'
  AND b.boatname = e.boatname
  AND r.race = e.race
```

```
SELECT boat,
       regatta_name
FROM entries
     NATURAL JOIN boats
     NATURAL JOIN races
WHERE boat = 'svuppe'
```

Source: Bulskov & Lassen (partial)

Competition 4

Competition 4

Find those makers that manufacture every type of printer.

Do *not* rely on the fact that there are only two types of printers in the given instance.

Competition 4

Find those makers that manufacture every type of printer.

Do *not* rely on the fact that there are only two types of printers in the given instance.

```
FORALL ptype. (EXISTS m, c, p. Printer(m, c, ptype, p)) IMPLIES
   (EXISTS m, t, c, p. Product(maker, m, t) AND Printer(m, c, ptype, p))
```

Competition 4

Find those makers that manufacture every type of printer.

Do *not* rely on the fact that there are only two types of printers in the given instance.

```
FORALL ptype. (EXISTS m, c, p. Printer(m, c, ptype, p)) IMPLIES
   (EXISTS m, t, c, p. Product(maker, m, t) AND Printer(m, c, ptype, p))
```

And now in SQL

Competition 4

Find those makers that manufacture every type of printer.

Do *not* rely on the fact that there are only two types of printers in the given instance.

```
FORALL ptype. (EXISTS m, c, p. Printer(m, c, ptype, p)) IMPLIES
   (EXISTS m, t, c, p. Product(maker, m, t) AND Printer(m, c, ptype, p))
```

And now in SQL

Two independent evaluation criteria: **shortest** and **most efficient** query

(You may submit two different queries.)

Outer joins

- A special type of join operator that returns not only matching but also **non-matching** tuples

- Flavors:
  - **Left:** returns non-matching tuples from left relation
  - **Right:** returns non-matching tuples from right relation
  - **Full:** returns non-matching tuples from both relations

Source: Ramakrishnan & Gehrke (partial)

List the sid’s of all sailors; for each sailor with reservations, list also the bid’s of boats she reserved

```
SELECT DISTINCT S.sid, R.bid
FROM   Sailors S
       LEFT OUTER JOIN
       Reserves R
       ON S.sid = R.sid
```

- Non-matching tuples paired with NULLs

Reserves

sid	bid	day
22	101	10.10
58	103	11.12

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Source: Ramakrishnan & Gehrke (partial)

List the sid's of all sailors; for each sailor with reservations, list also the bid's of boats she reserved

```
SELECT DISTINCT S.sid, R.bid
FROM   Sailors S
       LEFT OUTER JOIN
       Reserves R
       ON S.sid = R.sid
```

- Non-matching tuples paired with NULLs

sid	bid
22	101
58	103
31	NULL

Reserves

sid	bid	day
22	101	10.10
58	103	11.12

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Source: Ramakrishnan & Gehrke (partial)

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
```

Reserves

sid	bid	day
22	101	10.10
58	103	11.12

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Source: Ramakrishnan & Gehrke (partial)

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
```

Reserves

sid	bid	day
22	101	10.10
58	103	11.12

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

sid
22
31

Source: Ramakrishnan & Gehrke (partial)

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.

Source: Ramakrishnan & Gehrke

Queries and Null Values

- What if S.Age is NULL?
  - S.Age > 25 returns 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
```

Source: Ramakrishnan & Gehrke

Queries and Null Values

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
```

- What if S.Age is NULL?
  - S.Age > 25 returns NULL!
- Implies a predicate can return 3 values
  - True, false, NULL
  - **Three-valued logic!**

Source: Ramakrishnan & Gehrke

Queries and Null Values

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
```

- 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)

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

Source: Ramakrishnan & Gehrke

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
```

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

Source: Ramakrishnan & Gehrke, Garcia-Molina et al.

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
```

- 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

Source: Ramakrishnan & Gehrke, Garcia-Molina et al.

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

Source: Ramakrishnan & Gehrke, Garcia-Molina et al.

Expressions and Strings

Sailors

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%'
```

- 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.

Expressions and Strings

Sailors

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%'
```

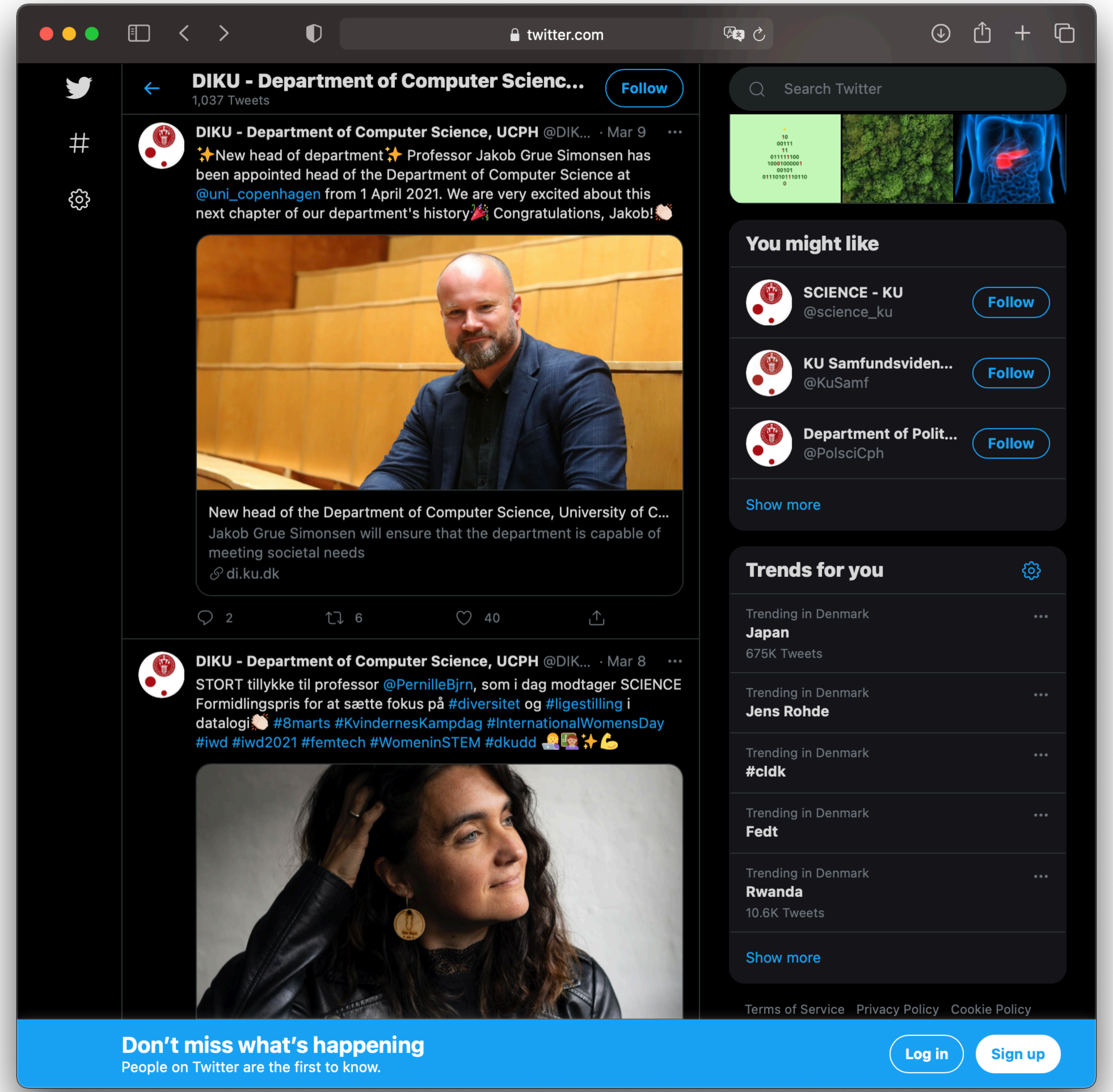
age	age5	age2
45	40	90
35	30	70
NULL	NULL	NULL

- 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



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
```

Source: Ramakrishnan & Gehrke (partial)

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
```

Source: Ramakrishnan & Gehrke (partial)

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)
```

Source: Ramakrishnan & Gehrke

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
```

Source: Ramakrishnan & Gehrke (partial)

R

a	b
1	2
3	4

S

b	c
2	5
2	6

And How About These?

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

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

Source: Bulskov & Lassen (partial)

R

a	b
1	2
3	4

S

b	c
2	5
2	6

And How About These?

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

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

a
1
1

Source: Bulskov & Lassen (partial)

And How About These?

R

a	b
1	2
3	4

S

b	c
2	5
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

Source: Bulskov & Lassen (partial)

More on 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')
```

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

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)
```

Source: Ramakrishnan & Gehrke (partial)

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, ... , 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
```

Source: Ramakrishnan & Gehrke (partial)

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
  - **Remaining tuples are partitioned into groups by the value of the attributes in grouping-list**
  - **One answer tuple is generated per group**
- 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

Source: Ramakrishnan & Gehrke (partial)

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

Source: Ramakrishnan & Gehrke (partial)

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

Source: Ramakrishnan & Gehrke (partial)

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
```

Source: Ramakrishnan & Gehrke (partial)

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
```

Source: Ramakrishnan & Gehrke (partial)

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 >= 18
for each rating level with at least 2 <u>such</u> sailors

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 >= 18
for each rating level 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
```

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
  - 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

- Recall: Does not imply query will actually be evaluated this way!

Source: Ramakrishnan & Gehrke

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)

Source: Ramakrishnan & Gehrke (partial)

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

Source: Ramakrishnan & Gehrke (partial)

Find the age of the youngest sailor with age $\geq$ 18, for each rating with at least 2 <u>such</u> 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

Source: Ramakrishnan & Gehrke (partial)

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
```

- ORDER BY can only appear in top-most query
  - Otherwise results are unordered!

Source: Ramakrishnan & Gehrke (partial)

What should we learn today?

- Formulate basic SQL queries, namely **select-project-join** queries
- Explain issues related to **duplicates** in SQL
- Formulate SQL statements to **insert**, **delete**, and **update** data
- Explain the semantics of the varied types of **joins** that can be formulated in SQL
- Explain the semantics of **NULL** in SQL and reason about expressions in a three-valued logic
- Identify the multiple constructions for **sub-queries** in SQL and explain their meaning
- Explain the semantics of **aggregation** operations and **grouping** in SQL
- Formulate queries that combine **joins**, **sub-queries**, **grouping**, & **aggregation**