Introduction to Database Systems IDBS - Spring 2024

Week 3 - Advanced SQL

Division
JOINS and NULL
Set Operations
Subqueries
Views

Readings: PDBM 7.3-7.4

Omar Shahbaz Khan

WHOAMI

- Name: Omar Shahbaz Khan

- Background:
  - 2013 - 2016: BSc, ITU (SWU)
  - 2016 - 2018: MSc, ITU (CS)
  - 2018 - 2022: PhD, ITU, Large Scale Interactive Learning for Multimedia
  - 2023 - Now: Postdoc, Reykjavik University, Multimedia Exploration in VR

- Research Interests:
  - Multimedia Analytics, Interactive Learning, Scalability, Databases, Big Data Management and Analytics

Running Example 1

Same schema as in week 2
Script on LearnIT: 03-coffees-schema/queries.sql

Coffees (name, manufacturer)
Coffeehouses (name, address, license)
Drinkers (name, address, phone)
Likes (drinker, coffee)
Sells (coffeehouses, coffee, price)
Frequents (drinker, coffeehouse)

Running Example 2

Sports Database same schema as in week 2 exercises
Script on LearnIT: 03-sports-schema/queries.sql

People (ID, name, gender, height)
Gender (gender, description)
Sports (ID, name, record)
Competitions (ID, place, held)
Results (peopleID, competitionID, sportID, result)

Last Time in IDBS...

Last Time...

- SELECT - FROM - WHERE
- JOIN
- ORDER BY, DISTINCT, Aggregations
- GROUP BY + HAVING

Last Time...

- SELECT - FROM - WHERE
- JOIN
- ORDER BY, DISTINCT, Aggregations
- GROUP BY + HAVING

```sql
SELECT name
  FROM Coffees;
```

```sql
SELECT name AS coffee
  FROM Coffees
 WHERE manufacturer LIKE 'Kopi%';
```

Last Time...

- SELECT - FROM - WHERE
- JOIN
- ORDER BY, DISTINCT, Aggregations
- GROUP BY + HAVING

```
SELECT f.drinker,
       f.coffeehouse,
       ch.address
  FROM Frequents f
  JOIN Coffeehouses ch ON
       f.coffeehouse = ch.name;
```

Last Time...

- SELECT - FROM - WHERE
- JOIN
- ORDER BY, DISTINCT, Aggregations
- GROUP BY + HAVING

```sql
SELECT *
  FROM Sells
 ORDER BY price ASC;

SELECT COUNT(*)
  FROM Sells;

SELECT COUNT(DISTINCT price)
  FROM Sells;
```

Last Time...

- SELECT - FROM - WHERE
- JOIN
- ORDER BY, DISTINCT, Aggregations
- GROUP BY + HAVING

```sql
SELECT coffee, MAX(price)
  FROM Sells
 GROUP BY coffee;
```

```sql
SELECT coffee, AVG(price)
  FROM Sells
 GROUP BY coffee
HAVING COUNT(coffeehouse) > 1;
```

Wake Up Task!

Sports DB

People (ID, name, gender, height)
Gender (gender, description)
Sports (ID, name, record)
Competitions (ID, place, held)
Results (peopleID, competitionID, sportID, result)

How many sports have the word “Jump” in them?

How many people have participated in more than 25 competitions?

This Time…

-- TODO

- Division
- JOINS & NULL
  - Natural Joins, Cross Joins, Self-Joins
  - NULL
  - Three-valued logic
  - Operators, Outer Joins
- Set Operations
  - UNION, INTERSECT, EXCEPT
- Subqueries (Nested Queries)
  - =, IN, EXISTS, ALL, ANY
- Views: Queries as subroutines

/

What is Division?

- R1 / R2 = tuples of R1 associated with all tuples of R2
- Find the students who have taken **all** courses in a program
- Find the airlines who land at **all** airports in a country/continent/the world
- Find the coffeehouses that sell **all** existing coffees

Division with Counting

- We can write *division* using GROUP BY, HAVING and a COUNT subquery

- Find the coffeehouses that sell **all** existing coffees

```
SELECT coffeehouse
  FROM Sells
 GROUP BY coffeehouse
HAVING COUNT(coffee) = (
  SELECT COUNT(*)
    FROM Coffees);
```

2. For each coffeehouse, return it only if it sells that many

1. Count the number of coffees

Coffees (name, manufacturer)
Coffeehouses (name, address, license)
Drinkers (name, address, phone)
Likes (drinker, coffee)
Sells (coffeehouses, coffee, price)
Frequents (drinker, coffeehouse)

Practice

- Names of drinkers who frequent all coffeehouses

```
SELECT drinker
  FROM Frequents
 GROUP BY drinker
HAVING COUNT(coffeehouse) = (
         SELECT COUNT(*)
           FROM Coffeehouses);
```

Coffees (name, manufacturer)
Coffeehouses (name, address, license)
Drinkers (name, address, phone)
Likes (drinker, coffee)
Sells (coffeehouses, coffee, price)
Frequents (drinker, coffeehouse)

More Complete/Complex Example

- Exercise 2, #20
  Show the ID, name, record and worst results of all sports that have at least one result from every place where a competition has ever been held

```
SELECT s.id, s.name, s.record, MIN(r.result)
  FROM Sports s
  JOIN Results r ON s.id = r.sportId
  JOIN Competitions c ON
       r.competitionId = c.id
 GROUP BY s.id, s.name, s.record
HAVING COUNT(DISTINCT c.place) = (
        SELECT COUNT(DISTINCT c.place)
          FROM Competitions c);
```

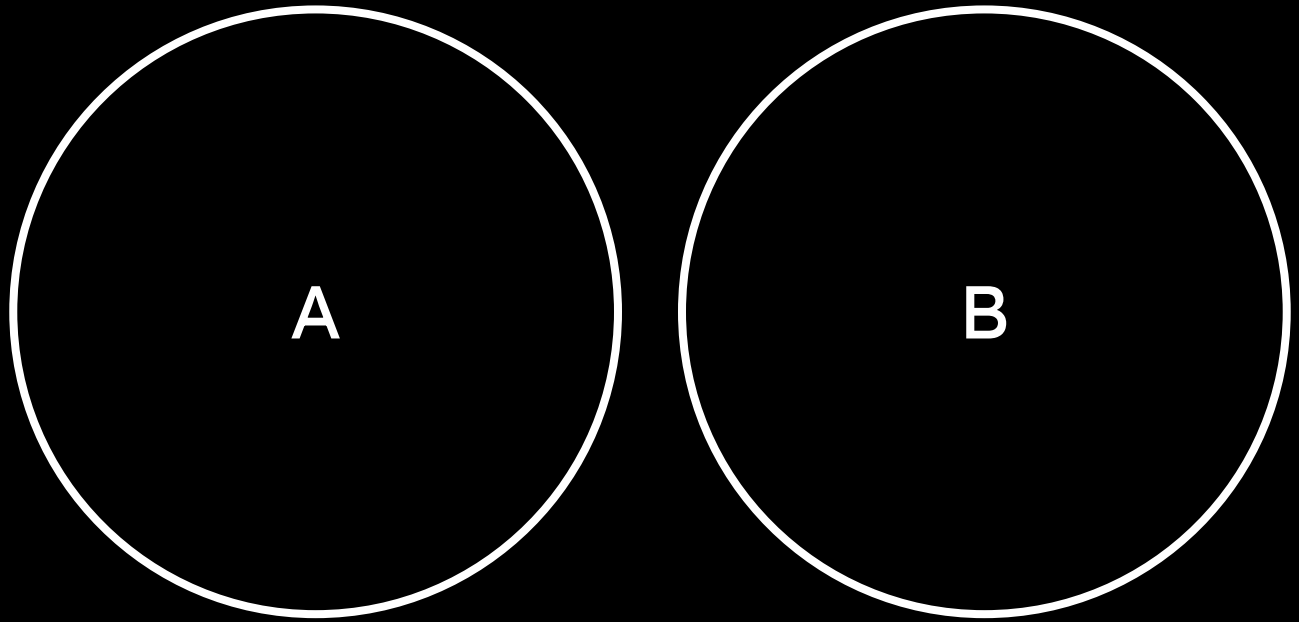
People (ID, name, gender, height)
Gender (gender, description)
Sports (ID, name, record)
Competitions (ID, place, held)
Results (peopleID, competitionID, sportID, result)

-- TODO

✓ Division

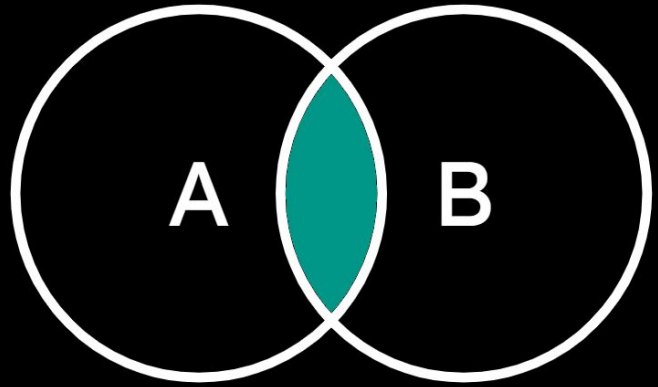
- JOINS & NULL
  - Natural Joins, Cross Joins, Self-Joins
  - NULL
  - Three-valued logic
  - Operators, Outer Joins

- Set Operations
  - UNION, INTERSECT, EXCEPT
- Subqueries (Nested Queries)
  - =, IN, EXISTS, ALL, ANY
- Views: Queries as subroutines


A
B

INNER JOIN

- Most joins are inner joins
- Combines the data of two relations based on a common column
- This is what we did last week
  - Following are some variants of INNER Joins




```
SELECT *
  FROM Drinkers d
  JOIN Frequents f ON
       d.name = f.drinker;
```

Natural Joins and Cross Joins (Product)

A NATURAL JOIN **B**:

- Assumes “=” on columns of same name
  - Removes duplicates
- Can be dangerous, not recommended!

```
SELECT *
  FROM Frequents
  NATURAL JOIN Likes;
```

A CROSS JOIN **B**:

- Removes no columns, has no join condition
  - Can simulate with an always true condition (ON 1 = 1)
  - Rarely (but not never) the right thing to do!

```
SELECT *
  FROM Likes l
  CROSS JOIN Drinkers d;
```

Self-Join

- From Coffees(name, manufacturer), find all pairs of coffees by the same manufacturer
  - Need a double loop to compare two coffee records
  - WHERE clause ensures no pairs with the same coffees, and alphabetical order

```
SELECT c1.name, c2.name,
       c1.manufacturer
  FROM Coffees c1
  JOIN Coffees c2 ON
       c1.manufacturer =
       c2.manufacturer
 WHERE c1.name < c2.name;
```

Self-Join: Example

- Show all coffees that are more expensive than some other coffee sold at the same coffeehouse

Coffees (name, manufacturer)
Coffeehouses (name, address, license)
Drinkers (name, address, phone)
Likes (drinker, coffee)
Sells (coffeehouses, coffee, price)
Frequents (drinker, coffeehouse)

1. Get all pairs:

```
SELECT *
  FROM Sells s1
  JOIN Sells s2
    ON s1.coffeehouse =
       s2.coffeehouse;
```

2. Remove unwanted pairs:

```
SELECT *
  FROM Sells s1
  JOIN Sells s2
    ON s1.coffeehouse =
       s2.coffeehouse
 WHERE s1.price > s2.price;
```

3. Remove duplicates and show only coffees:

```
SELECT DISTINCT s1.coffee
  FROM Sells s1
  JOIN Sells s2
    ON s1.coffeehouse =
       s2.coffeehouse
 WHERE s1.price > s2.price;
```

NULL Values

- Tuples in SQL relations can have NULL as a value for one or more components
- Meaning depends on context. Two Common cases:
  - Missing Value - e.g., we know Joe's Bar has some address, but we don't know what it is
    - "secret", "figure not available", "TBA", "impossible to calculate", "partly unknown", "uncertain", "pending"
  - Inapplicable - e.g., the value attribute spouse for an unmarried person
    - "undefined", "moot", "irrelevant", "none", "N/A"

Comparing NULL Values

- The logic conditions in SQL is 3-valued
  - TRUE, FALSE, UNKNOWN
- Comparing any value (including NULL itself) with NULL yields UNKNOWN
  - NULL != NULL
  - Must use IS NULL or IS NOT NULL
- A tuple is in a query answer iff the WHERE clause is TRUE (not FALSE or UNKNOWN)

Example

- Show the coffeehouses that sell a coffee at an unknown price

```
SELECT DISTINCT coffeehouse
  FROM Sells
 WHERE price IS NULL;
```

- Show the name of all coffees that are sold at a known price

```
SELECT DISTINCT coffee
  FROM Sells
 WHERE price IS NOT NULL;
```

Coffees (name, manufacturer)
Coffeehouses (name, address, license)
Drinkers (name, address, phone)
Likes (drinker, coffee)
Sells (coffeehouses, coffee, price)
Frequents (drinker, coffeehouse)

Three-Valued Truth Tables

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

=	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	TRUE	NULL
NULL	NULL	NULL	NULL

NULL vs Aggregation

- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column

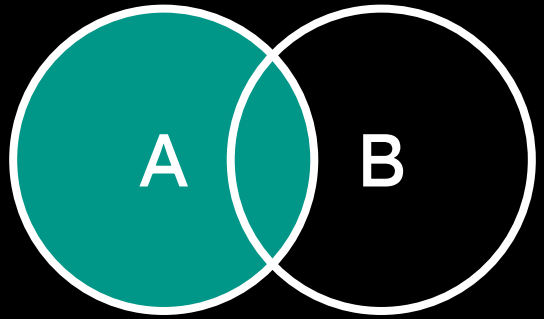
- But if there are no non-NULL values in a column then the result of the aggregation is NULL
    - Exception: COUNT of an empty set is 0

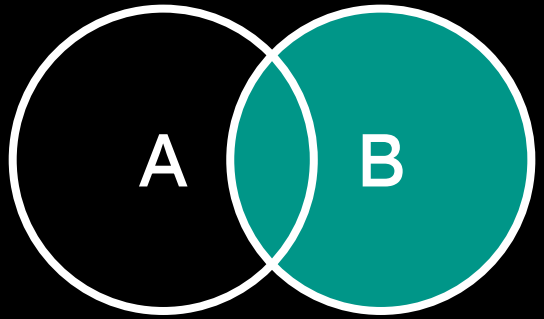
Outer Joins

- Sometimes a coffee is sold nowhere, but we want it in our results
- ***A*** OUTER JOIN ***B*** is the core of an outer join expression
- It is modified by LEFT, RIGHT, or FULL before OUTER
  - LEFT = pad dangling tuples of A with NULL
  - RIGHT = pad dangling tuples of B with NULL
  - FULL = pad both;

JOINS in a nutshell

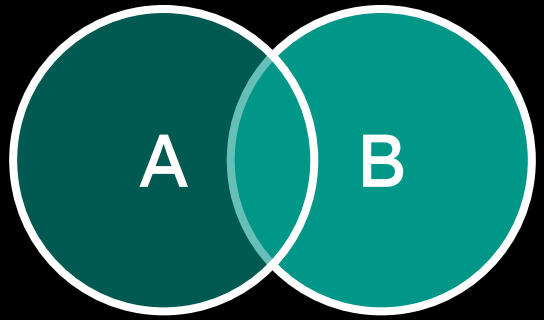
LEFT OUTER JOIN

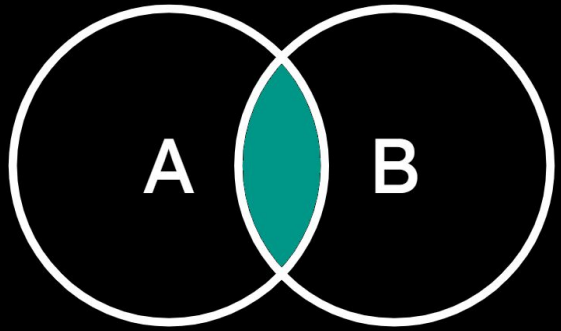




RIGHT OUTER JOIN

FULL OUTER JOIN





INNER JOIN

Example of an Outer Join

Suppose we add a new coffee:

```
INSERT INTO Coffees VALUES
('Bragakaffi', 'Ó.J. & Kaaber');
```

Coffees (name, manufacturer)
Coffeehouses (name, address, license)
Drinkers (name, address, phone)
Likes (drinker, coffee)
Sells (coffeehouses, coffee, price)
Frequents (drinker, coffeehouse)

- Show coffeehouses along with the manufacturers of the coffees they are selling.
  Also include manufacturers that do not sell any coffees.

```
SELECT s.coffeehouse,
       c.manufacturer
  FROM Sells s
 RIGHT OUTER JOIN Coffees c
    ON s.coffee = c.name;
```

Practice

- Show all drinkers and the coffees they like, but include drinkers that do not like any coffees

- Can you use OUTER JOIN to show *ONLY* drinkers who do not like any coffees?

Coffees (name, manufacturer)
Coffeehouses (name, address, license)
Drinkers (name, address, phone)
Likes (drinker, coffee)
Sells (coffeehouses, coffee, price)
Frequents (drinker, coffeehouse)

Practice

- Show all drinkers and the coffees they like, but include drinkers that do not like any coffees
- Can you use OUTER JOIN to show *ONLY* drinkers who do not like any coffees?

Coffees (name, manufacturer)
Coffeehouses (name, address, license)
Drinkers (name, address, phone)
Likes (drinker, coffee)
Sells (coffeehouses, coffee, price)
Frequents (drinker, coffeehouse)

WRONG!

```
SELECT *
  FROM Likes l
  JOIN Drinkers d ON d.name = l.drinker;
```

Correct!

```
SELECT *
  FROM Likes l
 RIGHT OUTER JOIN Drinkers d
    ON d.name = l.drinker;
```

Correct!

```
SELECT *
  FROM Drinkers d
  LEFT OUTER JOIN Likes l
    ON l.drinker = d.name;
```

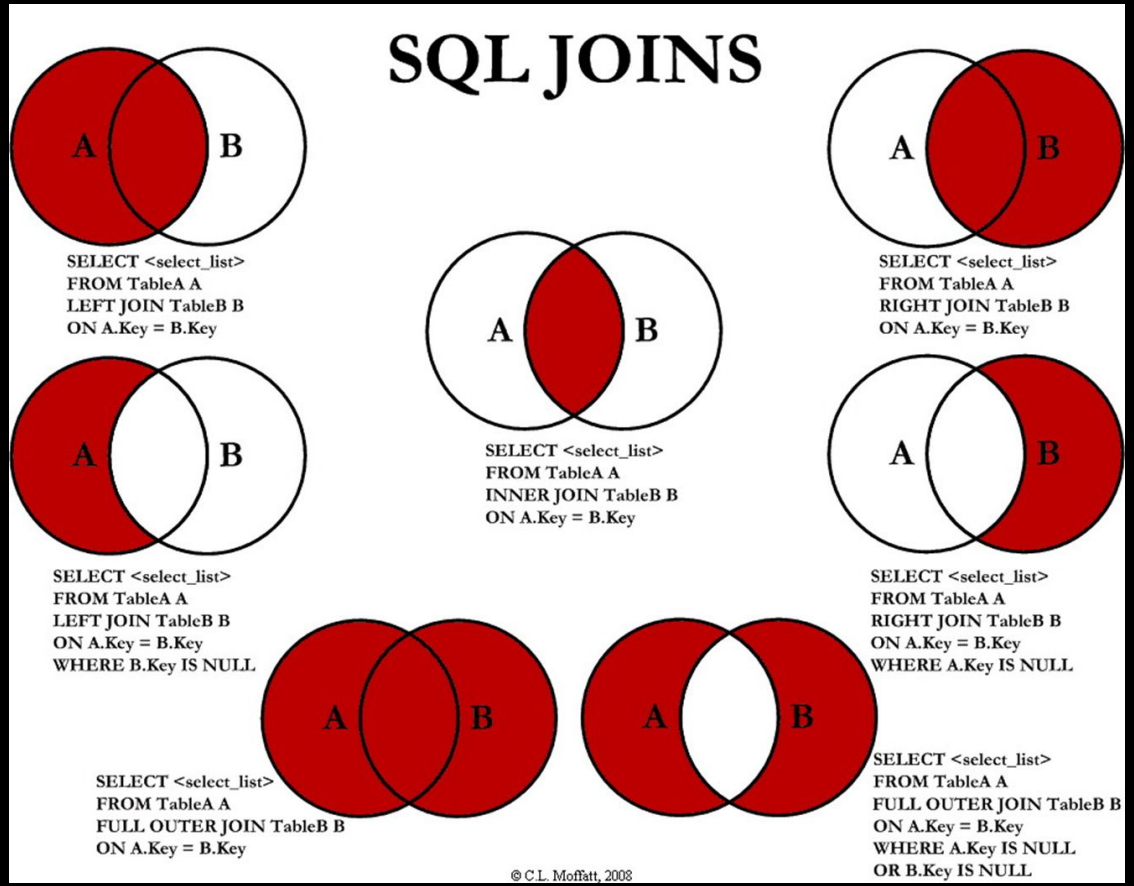
Practice

- Show all drinkers and the coffees they like, but include drinkers that do not like any coffees

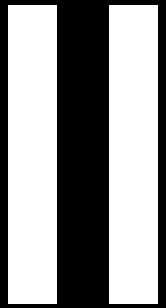
```
SELECT d.name
  FROM Drinkers d
  LEFT JOIN Likes l
    ON d.name = l.drinker
 WHERE l.drinker IS NULL;
```

- Can you use OUTER JOIN to show *ONLY* drinkers who do not like any coffees?

Coffees (name, manufacturer)
Coffeehouses (name, address, license)
Drinkers (name, address, phone)
Likes (drinker, coffee)
Sells (coffeehouses, coffee, price)
Frequents (drinker, coffeehouse)



https://www.codeproject.com/KB/database/Visual_SQL_Joins/Visual_SQL_JOINS_orig.jpg



-- TODO

- ✓ Division
- ✓ JOIN & NULL
  - ✓ Natural Joins, Cross Joins, Self-Joins
  - ✓ NULL
  - ✓ Three-valued logic
  - ✓ Operators, Outer Joins

- Set Operations
  - UNION, INTERSECT, EXCEPT
- Subqueries (Nested Queries)
  - =, IN, EXISTS, ALL, ANY
- Views: Queries as subroutines

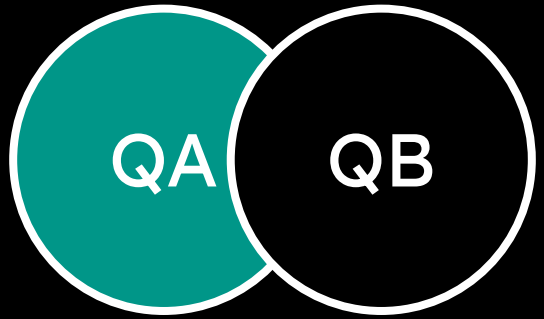
Set Queries

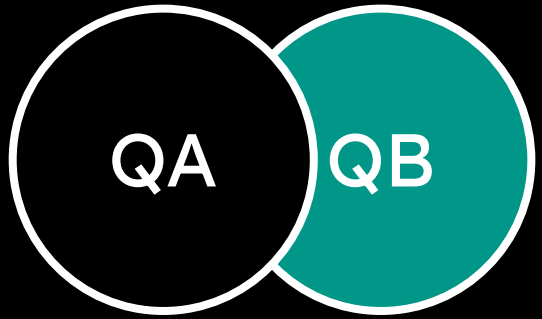
- Syntax:
  <Query 1>
  UNION/INTERSECT/EXCEPT
  <Query 2>

- Queries must be “union compatible” = results have matching schema:
  - Same number of attributes
  - Attributes *i* of both tables have same (matching type)

Set Operators in a nutshell

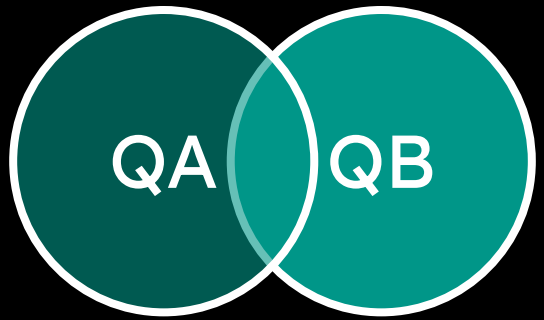
QA EXCEPT QB

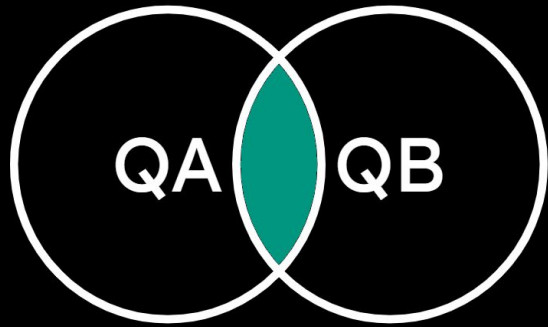




QB EXCEPT QA

QA UNION QB
QB UNION QA





QA INTERSECT QB
QB INTERSECT QA

Operators

- UNION
  Show all drinkers that like “Kopi luwak” or live in “Amager”

- INTERSECT
  Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price

- EXCEPT
  Show all drinkers that frequents “Analog” but not those that like the coffee “Kopi luwak”.

Operators

- UNION
  Show all drinkers that like “Kopi luwak” or live in “Amager”
- INTERSECT
  Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price
- EXCEPT
  Show all drinkers that frequents “Analog” but not those that like the coffee “Kopi luwak”.

```
SELECT l.drinker
  FROM Likes l
 WHERE l.coffee = 'Kopi luwak'
 UNION
SELECT d.name
  FROM Drinkers d
 WHERE d.address = 'Amager';
```

Operators

- UNION
  Show all drinkers that like “Kopi luwak” or live in “Amager”

- INTERSECT
  Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price

- EXCEPT
  Show all drinkers that frequents “Analog” but not those that like the coffee “Kopi luwak”.

```
SELECT c.name
  FROM Coffees c
 WHERE c.manufacturer = 'Marley Coffee'
INTERSECT
SELECT s.coffee
  FROM Sells s
 WHERE s.price IS NULL;
```

Operators

- UNION
  Show all drinkers that like “Kopi luwak” or live in “Amager”
- INTERSECT
  Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price
- EXCEPT
  Show all drinkers that frequents “Analog” but not those that like the coffee “Kopi luwak”.

```
SELECT f.drinker
  FROM Frequents f
 WHERE f.coffeehouse = 'Analog'
EXCEPT
SELECT l.drinker
  FROM Likes l
 WHERE l.coffee = 'Kopi luwak';
```

Duplicates

- Set operators remove duplicates

- To keep duplicates use:
  - *R* UNION ALL *S* - (0, 1, 2x5, 3x3, 4, 5)
  - *R* INTERSECT ALL *S* - (2x2, 1x3)
  - *R* EXCEPT ALL *S* - (1, 0x2, 1x3, 4)

- Most practical use:
  UNION ALL when *R* and *S* are known to be disjoint!

R	S
1	0
2	2
2	2
3	2
3	3
4	5

-- TODO

- ✓ Division
- ✓ JOIN & NULL
  - ✓ Natural Joins, Cross Joins, Self-Joins
  - ✓ NULL
  - ✓ Three-valued logic
  - ✓ Operators, Outer Joins
- ✓ Set Operations
  - ✓ UNION, INTERSECT, EXCEPT
- Subqueries (Nested Queries)
  - =, IN, EXISTS, ALL, ANY
- Views: Queries as subroutines

QUERY
QUERY
QUERY
QUERY
QUERY
QUERY

Subqueries

In SQL, a parenthesized SELECT-FROM-WHERE statement can be used as a value in many places, including WHERE and FROM clauses

```
SELECT *
  FROM R
 WHERE R.a <op>
       (SELECT S.b
          FROM S);
```

```
SELECT T.a
  FROM (SELECT R.a, S.c
          FROM R
          JOIN S
            ON R.a = S.b)
       AS T;
```

```
SELECT R.a
  FROM R
  JOIN (SELECT S.b, S.c
          FROM S) AS T
    ON R.a = T.b;
```

Single-Tuple (=)

- Subqueries return relations, but…
  If a subquery is guaranteed to produce one tuple, then it can be used as a value
  - Usually, the tuple has only one attribute
  - Implicit type cast
  - Run-time Error when >1 tuples or no tuples are returned

- Practical for:
  - Finding records with highest or lowest values
  - Division queries

```
SELECT coffee
  FROM Sells
 WHERE price =
       (SELECT MIN(price)
          FROM Sells);
```

```
SELECT s.coffeehouse
  FROM Sells s
 GROUP BY s.coffeehouse
HAVING COUNT(s.coffee) =
       (SELECT COUNT(*)
          FROM Coffees);
```

Practice

- Find the coffeehouses that serve some coffee for the same price Mocha charges for Blue Mountain

Coffees (name, manufacturer)
Coffeehouses (name, address, license)
Drinkers (name, address, phone)
Likes (drinker, coffee)
Sells (coffeehouses, coffee, price)
Frequents (drinker, coffeehouse)

Practice

- Find the coffeehouses that serve some coffee for the same price Mocha charges for Blue Mountain
- Using the Sells(coffeehouse, coffee, price) relation, this can be done in two queries:
  - Find the price Mocha charges for Blue Mountain
  - Find the coffeehouses that serve a coffee at that price
- Can it be done in a single query?

```sql
SELECT price
  FROM Sells
 WHERE coffeehouse = 'Mocha'
   AND coffee = 'Blue Mountain';

SELECT coffeehouse
  FROM Sells
 WHERE price = 300;
```

Practice

- Find the coffeehouses that serve some coffee for the same price Mocha charges for Blue Mountain
- Using the Sells(coffeehouse, coffee, price) relation, this can be done in two queries:
  - Find the price Mocha charges for Blue Mountain
  - Find the coffeehouses that serve a coffee at that price
- Can it be done in a single query?

```
SELECT coffeehouse
  FROM Sells
 WHERE price =
       (SELECT price
          FROM Sells
         WHERE coffeehouse = 'Mocha'
           AND coffee = 'Blue Mountain');
```

IN Operator

- Used to check if a tuple is IN a subquery's produced relation
  - The opposite can be achieved using NOT IN
- IN expressions can appear in WHERE clauses
- Example:
  Using Coffees(name, manufacturer) and Likes(drinker, coffee), find the name and manufacturer of each coffee that Johan likes

```
SELECT *
  FROM Coffees
 WHERE name IN (
         SELECT coffee
            FROM Likes
           WHERE drinker LIKE 'Johan%');
```

Example: NOT IN

- Show the name of all drinkers that does not frequent any coffeehouses

```
SELECT d.name
  FROM Drinkers d
 WHERE d.name NOT IN (
         SELECT f.drinker
           FROM Frequents f);
```

Set Operators: UNION

- Show all drinkers that like “Kopi luwak” or live in “Amager”

```sql
SELECT l.drinker
  FROM Likes l
 WHERE l.coffee = 'Kopi Luwak'
 UNION
SELECT d.name
  FROM Drinkers d
 WHERE d.address = 'Amager';
```

```sql
SELECT d.name
  FROM Drinkers d
 WHERE d.address = 'Amager'
    OR d.name IN (
        SELECT l.drinker
          FROM Likes l
         WHERE l.coffee = 'Kopi Luwak');
```

Set Operators: INTERSECT

- Show all coffees that are manufactured by “Marley Coffee” and sold at an unknown price

```sql
SELECT c.name
  FROM Coffees c
 WHERE c.manufacturer = 'Marley Coffee'
INTERSECT
SELECT s.coffee
  FROM Sells s
 WHERE s.price IS NULL;
```

```sql
SELECT c.name
  FROM Coffees c
 WHERE c.manufacturer = 'Marley Coffee'
   AND c.name IN (
       SELECT s.coffee
         FROM Sells s
        WHERE s.price IS NULL);
```

Set Operators: EXCEPT

- Show all drinkers that frequents “Analog” but not those that like the coffee “Kopi luwak”

```sql
SELECT f.drinker
  FROM Frequents f
 WHERE f.coffeehouse = 'Analog'
EXCEPT
SELECT l.drinker
  FROM Likes l
 WHERE l.coffee = 'Kopi luwak';
```

```sql
SELECT f.drinker
  FROM Frequents f
 WHERE f.coffeehouse = 'Analog'
   AND f.drinker NOT IN (
         SELECT l.drinker
           FROM Likes l
          WHERE l.coffee = 'Kopi luwak');
```

EXISTS Operator

- EXISTS (<subquery>)
  - Returns true if and only if a query result is not empty

- Example:
  Show the names of coffees that are liked by someone
  - Then show the opposite

```
SELECT c.name
  FROM Coffees c
 WHERE EXISTS (
        SELECT *
          FROM Likes l
         WHERE l.coffee = c.name);

SELECT c.name
  FROM Coffees c
 WHERE NOT EXISTS (
        SELECT *
          FROM Likes l
         WHERE l.coffee = c.name);
```

EXISTS Operator

- Example:
  From Coffees(name, manufacturer), find coffees of manufacturers that only sell a single coffee

```
SELECT c1.name
  FROM Coffees c1
 WHERE NOT EXISTS (
         SELECT *
           FROM Coffees c2
          WHERE c2.manufacturer =
                  c1.manufacturer
            AND c2.name <> c1.name);
```

Set of coffees with the same manufacturer as c1, but not the same name

Practice

Insert the following:

```
INSERT INTO Coffeehouses VALUES
('Kaffitar', 'Stroget', 'No question');
```

- Using NOT EXISTS, show the name of all coffeehouses that no one frequents

Practice

Insert the following:

```
INSERT INTO Coffeehouses VALUES
('Kaffitar', 'Stroget', 'No question');
```

- Using NOT EXISTS, show the name of all coffeehouses that no one frequents

```
SELECT ch.name
  FROM Coffeehouses ch
 WHERE NOT EXISTS (
         SELECT *
           FROM Frequents f
          WHERE f.coffeehouse = ch.name);
```

ANY Operator

- x <op> ANY (<subquery>)
  - Returns true iff at least one tuple in the subquery satisfies the comparison operator =, >, <, …
- Example:
  - For each coffeehouse, show all coffees that are more expensive than some other coffee sold at that coffeehouse

```
SELECT coffee
  FROM Sells
 WHERE price > ANY (
       SELECT price
         FROM Sells);
```

ALL Operator

- x <op> ALL (<subquery>)
  - Returns true iff for every tuple in the relation satisfies the comparison operator =, <, >, …
- Example:
  - From Sells(coffeehouse, coffee, price), find the coffee(s) sold for the highest price

```
SELECT coffee
  FROM Sells
 WHERE price >= ALL (
         SELECT price
           FROM Sells
          WHERE price IS NOT NULL);
```

FROM

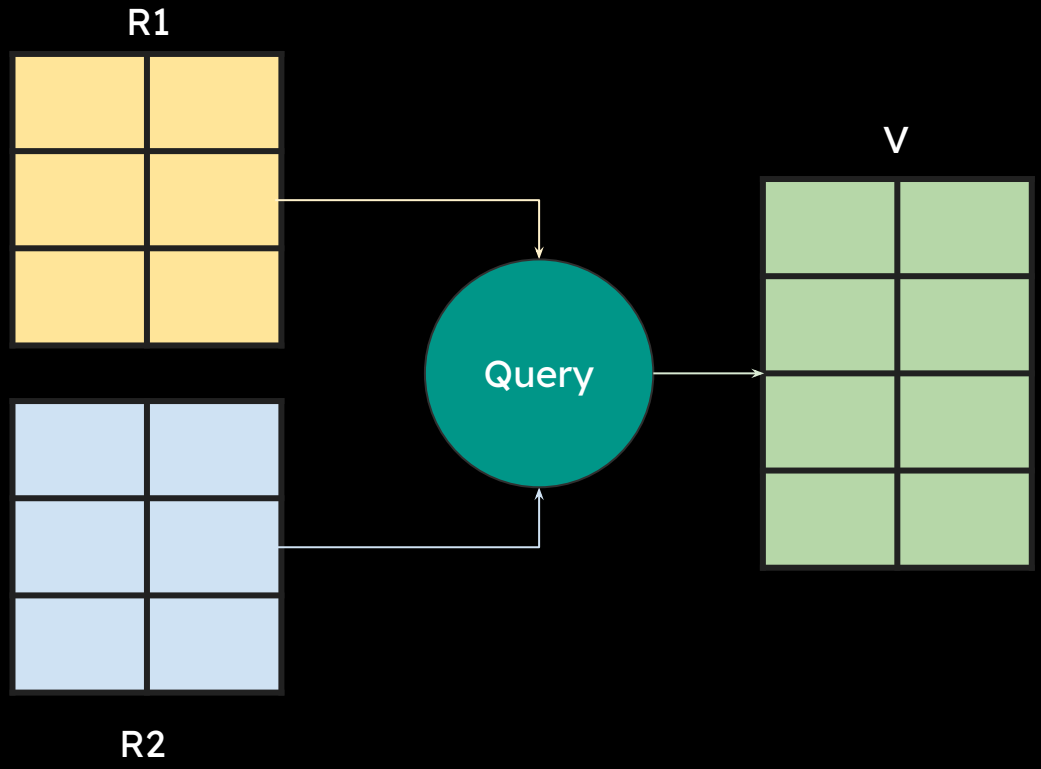
- Subqueries can also be used in FROM clauses
- Must use a tuple-variable to name tuples of the results

- Example:
  - Find the coffees liked by at least one person who frequents Mocha

```
SELECT DISTINCT l.coffee
  FROM Likes l
  JOIN (SELECT drinker
          FROM Frequents
         WHERE coffeehouse =
                'Mocha') md
       ON l.drinker = md.drinker;
```

-- TODO

- ✓ Division
- ✓ JOIN & NULL
  - ✓ Natural Joins, Cross Joins, Self-Joins
  - ✓ NULL
  - ✓ Three-valued logic
  - ✓ Operators, Outer Joins
- ✓ Set Operations
  - ✓ UNION, INTERSECT, EXCEPT
- ✓ Subqueries (Nested Queries)
  - ✓ =, IN, EXISTS, ALL, ANY
- Views: Queries as subroutines


R1
V
Query
R2

What is a View?

- A virtual table constructed from actual tables on the fly
  - Accessed in queries like any other table
  - Not materialized, constructed when accessed
  - Similar to a subroutine in ordinary programming
  - Is a schema element

```
CREATE VIEW Name (Columns,...)
AS
<SQL QUERY>
```

Example of View (Sports DB)

- For each result, we want:
  - the name of the athlete
  - the name of the sport
  - the percentage of the record achieved by the result (a result that is a record should therefore appear as 100; this column should be named “percentage”).

```
SELECT p.name, s.name,
ROUND(100*CAST((r.result/s.record) AS
numeric),0) AS percentage
  FROM People p
  JOIN Results r ON p.id = r.peopleid
  JOIN Sports s ON s.id = r.sportsid;
```

People (ID, name, gender, height)
Gender (gender, description)
Sports (ID, name, record)
Competitions (ID, place, held)
Results (peopleID, competitionID, sportID, result)

Example of View (Sports DB)

- For each result, we want:
  - the name of the athlete
  - the name of the sport
  - the percentage of the record achieved by the result (a result that is a record should therefore appear as 100; this column should be named “percentage”).

People (ID, name, gender, height)
Gender (gender, description)
Sports (ID, name, record)
Competitions (ID, place, held)
Results (peopleID, competitionID, sportID, result)

```
CREATE VIEW
      E2Q10 (pid, pname, sid, sname, percentage)
AS
SELECT p.id, p.name, s.id, s.name,
ROUND(100*CAST((r.result/s.record) AS
numeric),0) AS percentage
  FROM People p
  JOIN Results r ON p.id = r.peopleid
  JOIN Sports s ON s.id = r.sportid;


SELECT pname, sname, percentage
  FROM E2Q10;
```

Practice (Sports DB)

Create views for these queries:

- The ID, name and gender of all athletes who participated in the competition held in Hvide Sande in 2009.
- The name and gender of all people with last names that starts with a “J” and ends with a “sen”

People (ID, name, gender, height)
Gender (gender, description)
Sports (ID, name, record)
Competitions (ID, place, held)
Results (peopleID, competitionID, sportID, result)

```sql
SELECT DISTINCT p.id, p.name, g.description
  FROM People p
  JOIN Gender g ON p.gender = g.gender
  JOIN Results r ON p.id = r.peopleid
  JOIN Competitions c
    ON c.id = r.competitionid
 WHERE c.place = 'Hvide Sande'
   AND EXTRACT(YEAR FROM c.held) = 2009;

SELECT p.name, g.description
  FROM People p
  JOIN Gender g ON p.gender = g.gender
 WHERE p.name LIKE '% J%sen';
```

-- TODO -> DONE

- ✓ Division
- ✓ JOIN & NULL
  - ✓ Natural Joins, Cross Joins, Self-Joins
  - ✓ NULL
  - ✓ Three-valued logic
  - ✓ Operators, Outer Joins
- ✓ Set Operations
  - ✓ UNION, INTERSECT, EXCEPT
- ✓ Subqueries (Nested Queries)
  - ✓ =, IN, EXISTS, ALL, ANY
- ✓ Views: Queries as subroutines

Takeaways

SQL

SEQUEL

SQUEAL

SQUIRREL

One Block

- SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY

Many Blocks

- Multi Query
  - Connected by: UNION, INTERCEPT, EXCEPT
- Subqueries in FROM and WHERE
  - Blocks connected by: =, IN, EXISTS, ANY, ALL
  - Opposites (usually) with NOT
  - BUT: Prefer to use Joins when possible
- Division is one important type of queries with >1 query block!

Next Time in IDBS...

Introduction to Database Systems IDBS - Spring 2024

Week 4 - SQL and Python

Functions
Triggers
SQL and DBMS in Python
Transactions

Readings: PDBM 9.2, 14.1, 14.2.1 and 14.5

Omar Shahbaz Khan