CS2102: Database Systems

SQL (Part 3)

Announcements

Announcement

Midterm

- **Week 7:** Monday, 27 February 2023 (*12:30 - 13:30*)
- MPSH 2A & MPSH 2B
- Seating arrangement will be sent nearer to the assessment time
- **Softwares**
  - **Examplify:** for the assessment question and *(most)* answers
    - Practice using **Practice Exam - Non-Secure Block Internet**
    - Practice accessing the attached files
      - There can be attached file to the *assessment (i.e., global)*
      - There can be attached file to the *question (i.e. local)*
  - **PostgreSQL:** for testing your answer before submitting to Examplify
    - We recommend **psql** but you may use other software to access PostgreSQL too
    - Please check compatibility with Examplify
    - We will be using the same schema as in **Assignment #1** so please get yourself familiar with it

Announcement

Midterm (Rough Timeline)

1. **Download** the assessment before coming to MPSH
   - You will not be given additional time to download
   - Only download on the machine you intend to use for the assessment *(only 1 download)*

2. You can enter the venue by 12:15 *(please sit down on your assigned seat)*
   - There are seats dedicated for charging, these are limited so please be courteous

3. Open **psql** first *(may require some initial connection that may be blocked by Examplify once assessment starts)*
4. Open **Examplify** and login using your ID
   - Choose **National University of Singapore - APAC (nus)**

5. We will release the password around 12:29
   - Type in the password and start the assessment promptly
   - Except for technical error, no additional time will be granted for students who started late
     - Some consideration is given for slow load as long as you click start as soon as the password is released

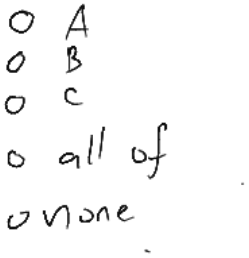
Announcement

Midterm (End of Assessment)

- You have until the end of the timer to do your work
  - Unless you started late for non-technical reason
- You should wait on the **Green Screen** of successful submission
  - Do not close Examplify until you have seen this message
- Please wait in your seat until you are released
  - We will need to tally all the submissions

Announcement

Midterm (End of Assessment)



- **Topic**
  - L01 to L06 *(yes, this lecture included)*
- **Types of *(possible)* Questions**
  - ***MCQ:*** only one correct answer *(choose the most appropriate)*
  - ***MRQ:*** may have more than one correct answers *(choose all correct answers)*
  - ***FITB:*** fill in the blank
  - ***SQL:*** write your query using the `VIEW` provided in the question `.pdf`
    - You should check using PostgreSQL that your code runs
      - Code that cannot be run may receive an immediate 0
  - ***ERD:*** draw the ERD in the paper provided and submit them to one of the invigilators at the end
    - If this question appear, the actual answer will not be on Examplify
- Please do not forget to take your belonging with you

Announcement

Assignment #1

- **Week 6.5:** Saturday, 25 February 2023 (*12:00* Noon)
- Submit only the `.sql` file containing the `VIEW` *(DDL and Data file will not be used)*
- ***Late Submission Penalty***
  - 2 marks *(out of 5)* will be deducted
  - Late submission more than 1 day will not be accepted

- Solution will be posted on **Sunday, 26 February 2023** to facilitate revision for midterm

Recap

$$\delta_c(R - S) \approx \delta_c(R) - \delta_c(S)$$

$$c := \frac{a}{b} < 10$$

$R(a)$

$S(b)$

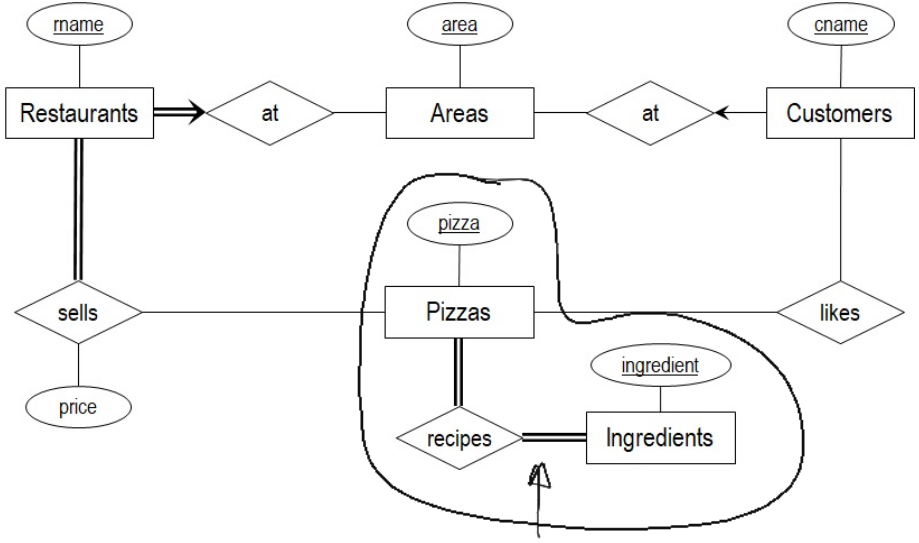
$$\delta_c(R - S) \equiv \delta_c(R) - S$$

Recap

Database
- ER Diagram
- Schema
- Sample

Example Database

ER Diagram



[#]This is only an *approximate* to help visualize the schema.

Recap

Database
- ER Diagram
- Schema
- Sample

Example Database

Schema

Relations

- *Restaurants(<u>rname</u>, area)*
- *Customers(<u>cname</u>, area)*
- *Pizzas(<u>pizza</u>)*
- *Ingredients(<u>ingredient</u>)*
- *Sells(<u>rname</u>, <u>pizza</u>, price)*
- *Likes(<u>cname</u>, <u>pizza</u>)*
- *Recipes(<u>pizza</u>, <u>ingredient</u>)*

Foreign Key

- *(Sells.rname)* ⇝ *(Restaurants.rname)*
- *(Sells.pizza)* ⇝ *(Pizzas.pizza)*
- *(Likes.cname)* ⇝ *(Customers.cname)*
- *(Likes.pizza)* ⇝ *(Pizzas.pizza)*
- *(Recipes.pizza)* ⇝ *(Pizzas.pizza)*
- *(Recipes.ingredient)* ⇝ *(Ingredients.ingredient)*

SQL (L06.sql)

```
CREATE TABLE Pizzas (
  pizza       VARCHAR NOT NULL PRIMARY KEY
);
```

Recap

Database
- ER Diagram
- Schema
- Sample

Example Database

Sample Database

Table "Sells"

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Gambino Oven	Hawaiian	0
Lorenzo Tavern	Funghi	23
Mammas Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

Table "Restaurants"

rname	area
Corleone Corner	North
Gambino Oven	Central
Lorenzo Tavern	Central
Mammas Place	South
Pizza King	East

Table "Customers"

cname	area
Homer	West
Lisa	South
Maggie	East
Moe	Central
Ralph	Central
Alice	Central
Bob	Central
Willie	North
Adi	NULL
Yoga	NULL

Table "Pizzas"

pizza
Diavola
Funghi
Hawaiian
Margherita
Marinara
Siciliana

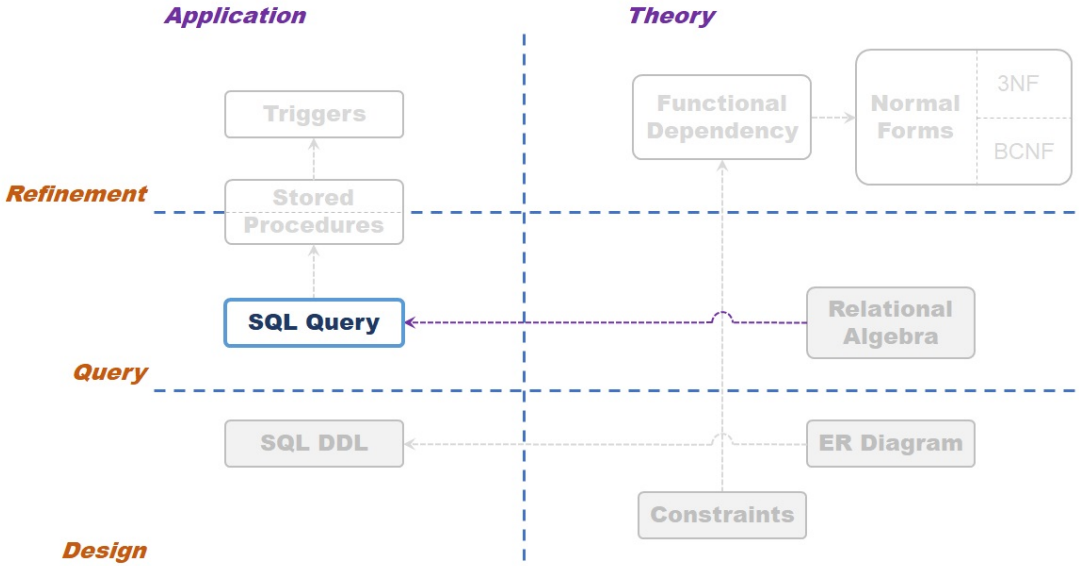
Table "Likes"

cname	pizza
Homer	Hawaiian
Homer	Margherita
Lisa	Funghi
Maggie	Funghi
Moe	Funghi
Moe	Siciliana
Ralph	Diavola
Alice	Siciliana
Alice	Funghi
Bob	Siciliana

Table "Recipes"

pizza	ingredient
Diavola	Cheese
Diavola	Chilli
Diavola	Salami
Funghi	Ham
Funghi	Mushroom
Hawaiian	Ham
Hawaiian	Pineapple
Margherita	Cheese
Margherita	Tomato
Marinara	Seafood
Marinara	Chilli
Siciliana	Anchovies
Siciliana	Caspers
Siciliana	Cheese

Roadmap


Application
Theory
Triggers
Functional Dependency
Normal Forms
3NF
BCNF
Refinement
Stored Procedures
SQL Query
Relational Algebra
Query
SQL DDL
ER Diagram
Constraints
Design

Roadmap

Overview

Overview

Common SQL Constructs

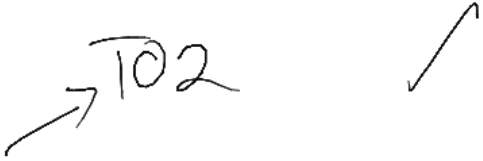
- Aggregation
- Grouping
- Conditional Expressions

Structuring Queries

- Common Table Expressions
- Views

Extended Concepts

- Universal Quantification
- Recursive Queries



Common SQL Constructs

Common SQL Constructs

Aggregation

Aggregate Functions

- Aggregate functions compute a ***single value*** from a set of tuples
  - Example: `MIN()`, `MAX()`, `AVG()`, `COUNT()`, `SUM()`

> **Question**
>
> Find the lowest and highest price pizza among all the restaurants that sells at least one pizza.

```
SELECT MIN(price) AS lowest,
       MAX(price) AS highest
FROM   Sells;
```

lowest	highest
0	25

Common SQL Constructs

Aggregation

Interpretation

> Let *R* be a non-empty relation with attribute *A*.

Query	Interpretation	Result
`SELECT MIN(A) FROM R;`	Minimum *non-NULL* value in *A*	0
`SELECT MAX(A) FROM R;`	Maximum *non-NULL* value in *A*	42
`SELECT AVG(A) FROM R;`	Average of *non-NULL* values in *A*	12 *(i.e., 48 / 4)*
`SELECT SUM(A) FROM R;`	Sum of *non-NULL* values in *A*	48
`SELECT COUNT(A) FROM R;`	Count of *non-NULL* values in *A*	4
`SELECT COUNT(*) FROM R;`	Count of rows in *A*	5
`SELECT AVG(DISTINCT A) FROM R;`	Average of ***distinct*** *non-NULL* value in *A*	15
`SELECT SUM(DISTINCT A) FROM R;`	Sum of ***distinct*** *non-NULL* value in *A*	45
`SELECT COUNT(DISTINCT A) FROM R;`	Count of ***distinct*** *non-NULL* value in *A*	3

...	A	...
...	3	...
...	42	...
...	NULL	...
...	0	...
...	3	...

Common SQL Constructs

...	B	...

...	A	...
...	NULL	...
...	NULL	...
⋮	⋮	⋮

Aggregation

Interpretation

> Let *R* be an empty relation with attribute *B*.

Query	Result
`SELECT MIN(B) FROM R;`	NULL
`SELECT MAX(B) FROM R;`	NULL
`SELECT AVG(B) FROM R;`	NULL
`SELECT SUM(B) FROM R;`	NULL
`SELECT COUNT(B) FROM R;`	0
`SELECT COUNT(*) FROM R;`	0

> Let *S* be a non-empty relation with *n*-rows and with attribute *A* that only has **NULL** values.

Query	Result
`SELECT MIN(A) FROM S;`	NULL
`SELECT MAX(A) FROM S;`	NULL
`SELECT AVG(A) FROM S;`	NULL
`SELECT SUM(A) FROM S;`	NULL
`SELECT COUNT(A) FROM S;`	0
`SELECT COUNT(*) FROM S;`	*n*

Common SQL Constructs

Aggregation

Function Signatures

Function	Input Type	Output Type
`MIN`	*any comparable type*	*same as input*
`MAX`	*any comparable type*	*same as input*
`SUM`	Numeric data *(e.g., `INT`, `BIGINT`, `REAL`, etc)*	`SUM(INT)` → `BIGINT`; `SUM(REAL)` → `REAL`
`COUNT`	*any data*	`BIGINT`

Common SQL Constructs

Grouping

Grouping and Aggregation

- So far, application of aggregate functions are over *all* tuples of a relation
  - Resulting relation only has *one* tuple

GROUP BY

Syntax

```
GROUP BY attr1, attr2, ...
```

Notes:

- Logical partition of relation into groups based on values for specified attributes
- In principle, always applied together with aggregation
  (`GROUP BY` without aggregation is valid but typically not meaningful)
- Application of aggregation functions are now over *each* group
  - One result tuple for each group

Common SQL Constructs

Grouping

Logical Partitioning

> **Question**
>
> For each restaurant, find the lowest and highest price pizza.

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Gambino Oven	Siciliana	0
Lorenzo Tavern	Funghi	23
Mammas Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

Query

```
SELECT rname,
       MIN(price),
       MAX(price)
FROM   Sells
GROUP BY rname;
```

rname	min	max
Lorenzo Tavern	23	23
Pizza King	17	21
Mammas Place	22	22
Gambino Oven	0	16
Corleone Corner	19	25

Common SQL Constructs

Grouping

Defining Groups

- Given `GROUP BY a1, a2, ..., an` two tuple *t* and *t'* belong to the same group if

```
(t.a1 IS NOT DISTINCT FROM t'.a1) AND
(t.a2 IS NOT DISTINCT FROM t'.a2) AND
   ... AND ...
(t.an IS NOT DISTINCT FROM t'.an)
```

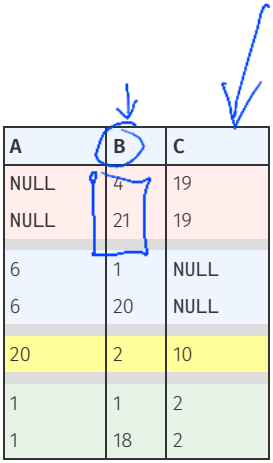
evaluates to `True`

Example

Table *R(A,B,C)*

A	B	C
NULL	4	19
6	1	NULL
20	2	10
1	1	2
1	18	2
NULL	21	19
6	20	NULL

```
SELECT *
FROM R
GROUP BY A,C;
```



A	B	C
NULL	4	19
NULL	21	19
6	1	NULL
6	20	NULL
20	2	10
1	1	2
1	18	2

Notes

This is a *logical* partitioning, it is possible that there is no actual reordering performed.

Common SQL Constructs

Grouping

Quiz #1

What is the problem with the invalid query?

Restrictions to SELECT Clause

Restriction

If column A_i of table R appears in the `SELECT` clause, one of the following conditions must hold

1. A_i appears in the `GROUP BY` clause
2. A_i appears as input of an aggregation function in the `SELECT` clause
3. The primary key ~~or a candidate key~~* of R appears in the `GROUP BY` clause

Example (Valid)

```
SELECT rname, SUM(price)
FROM   Sells
GROUP BY rname;
```

Example (Invalid)

```
SELECT rname, pizza, SUM(price)
FROM   Sells
GROUP BY rname;
```

*This is valid in standard SQL but not supported in PostgreSQL. In this module, we follow PostgreSQL's tighter restriction.

Common SQL Constructs

Grouping

Grouping Over Primary Key

> Consider the table "Countries" on the right.

```sql
CREATE TABLE Countries (
  iso2        CHAR(2) PRIMARY KEY,
  name        VARCHAR(255) UNIQUE,
  population  INTEGER,
  gdp         NUMERIC,
  continent   VARCHAR(255)
);
```

> **Quiz #2**
>
> What is the "problem" with the *valid* query?

Valid Query

```sql
SELECT name, population, COUNT(*)
FROM   Countries
GROUP BY iso2;
```

Invalid Query

```sql
SELECT name, population, COUNT(*)
FROM   Countries
GROUP BY name;
```

[#]The invalid query is ***valid*** in SQL standard but ***invalid*** in PostgreSQL even if we add `NOT NULL` constraint on `name`.

Common SQL Constructs

Grouping

> **Quiz #3**
>
> Will it be valid if we do `SELECT R.rname, C.area ...` instead of `SELECT R.rname, R.area ...`?

Valid Query

```
SELECT R.rname, R.area,
       COUNT (C.cname)
FROM   Restaurants R, Customers C
WHERE  R.area = C.area
GROUP BY R.rname;
```

Invalid Query

```
SELECT R.rname, C.cname,
       COUNT (C.cname)
FROM   Restaurants R, Customers C
WHERE  R.area = C.area
GROUP BY R.rname;
```

rname	area	cname	area
Corleone Corner	North	Willie	North
Gambino Oven	Central	Bob	Central
Gambino Oven	Central	Alice	Central
Gambino Oven	Central	Ralph	Central
Gambino Oven	Central	Moe	Central
⋮	⋮	⋮	⋮
Pizza King	East	Maggie	East

Common SQL Constructs

avg > overall avg

HAVING Clause

Where: remove the row where 'avg <= overall'.

but then the avg changes

⇒ infinite reasoning

Syntax

```
GROUP BY attr1, attr2, ...
HAVING <condition>
```

Notes:

- Conditions check for each group defined by `GROUP BY` clause
- `HAVING` clause cannot be used without `GROUP BY` clause
- Conditions typically involve aggregate functions

Having: filter on groups

Common SQL Constructs

HAVING Clause

Example

Question

Find all restaurants and the number of pizza they sells such that the retaurant sells more than 1 pizza.

Query

```
SELECT rname, COUNT(pizza)
FROM   Sells
GROUP BY rname
HAVING COUNT(pizza) > 1;
```

Result

rname	count
Pizza King	2
Gambino Oven	2
Corleone Corner	3

"where" cannot be together with aggregation.

Common SQL Constructs

HAVING Clause

Exercise #1

min(pizza)

max(pizza)

Question

Find all restaurants that sells any pizza cheaper than any pizza sold by Lorenzo Tavern.

Query

```
1  SELECT S.rname
2  FROM   Sells S
3  GROUP BY S.rname
4  HAVING MIN(price) < (
5      SELECT MAX(price)
6      FROM   Sells S1
7      WHERE  rname = 'Lorenzo Tavern'
8    );
```

Result

rname
Pizza King
Mammas Place
Gambino Oven
Corleone Corner

Common SQL Constructs

HAVING Clause

Restrictions to HAVING Clause

Restriction

If column A_i of table R appears in the `HAVING` clause, one of the following conditions must hold

1. A_i appears in the `GROUP BY` clause
2. A_i appears as input of an aggregation function in the `HAVING` clause
3. The primary key ~~or a candidate key~~* of R appears in the `GROUP BY` clause

Valid

```
SELECT rname, COUNT(*) FROM Sells
GROUP BY rname HAVING AVG(price) > 20;
```

```
SELECT rname, COUNT(*) FROM Sells
GROUP BY rname, pizza HAVING price > 20;
```

where

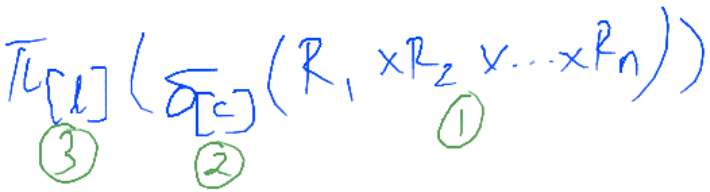
Invalid

```
SELECT rname, COUNT(*) FROM Sells
GROUP BY rname HAVING price > 20;
```

Quiz #4

What is the result of the query above?

Common SQL Constructs



HAVING Clause

Conceptual Evaluation of Queries

`FROM`	Compute cross-product of all tables in FROM clause
`WHERE`	Filter (*i.e., keep*) tuples that evaluate to true on the WHERE condition(s)
`GROUP BY`	Partition table into groups w.r.t. the grouping attribute(s)
`HAVING`	Filter (*i.e., keep*) groups that evaluate to true on the HAVING condition(s)
`SELECT`	Remove all attribute(s) not specified in SELECT clause (*& remove duplicate if DISTINCT*)
`ORDER BY`	Sort tables based on specified attribute(s)
`LIMIT/OFFSET`	Filter (*i.e., keep*) tuples based on their order in the table

Common SQL Constructs

Aggregation
Grouping
HAVING
Conditional
- CASE
- COALESCE
- NULLIF

Conditional Expressions

CASE Expression

Syntax

```
CASE
  WHEN <condition_1> THEN <result_1>
  WHEN <condition_2> THEN <result_2>
    ⋮
  WHEN <condition_n> THEN <result_n>
  ELSE result_0
END
```

```
CASE <expression>
  WHEN <value_1> THEN <result_1>
  WHEN <value_2> THEN <result_2>
    ⋮
  WHEN <value_n> THEN <result_n>
  ELSE <result_0>
END
```

Notes:

- Only one of the results will be returned
- If `<expression>` is specified, similar to `<condition>` being defined as `<expression> = <value>`
- `ELSE` is *optional*
  - Return `NULL` if no `ELSE` is specified and no condition matched

Common SQL Constructs

Conditional Expressions

CASE Expression

> **Question**
>
> For each restaurant, we categorize the restaurant by the average price of their pizza.

Category	Price
Expensive	price ≥ 23
Reasonable	23 > price ≥ 18
Cheap	18 > price

Query

```
SELECT rname, CASE
    WHEN AVG(price) >= 23 THEN 'Expensive'
    WHEN AVG(price) >= 18 THEN 'Reasonable'
    ELSE 'Cheap'
  END
FROM Sells
GROUP BY rname;
```

Table

rname	case
Lorenzo Tavern	Expensive
Pizza King	Reasonable
Mammas Place	Reasonable
Gambino Oven	Cheap
Corleone Corner	Reasonable

Common SQL Constructs

Aggregation
Grouping
HAVING
Conditional
- CASE
- COALESCE
- NULLIF

Conditional Expressions

COALESCE Expression

Syntax

```
COALESCE(<value_1>, <value_2>, <value_3>, ...)
```

Notes:

- Returns the first non-`NULL` value in the list of input arguments *(order of `<value>` matters!)*
- Returns `NULL` if all values in the list of input arguments are `NULL`

Equivalence

```
COALESCE(<value_1>,
         <value_2>,
         <value_3>)
```

≡

```
CASE
  WHEN <value_1> IS NOT NULL THEN <value_1>
  WHEN <value_2> IS NOT NULL THEN <value_2>
  ELSE <value_3>
END
```

Common SQL Constructs

Aggregation
Grouping
HAVING
Conditional
- CASE
- COALESCE
- NULLIF

Conditional Expressions

COALESCE Expression

Scenario

A student may attempt a quiz *up to* 3 times until they pass. Each potential attempt is recorded as follows:

- On *passing* the value is recorded as `'pass'`; cannot attempt another
- On *failing* the value is recorded as `'fail'`; may attempt another
- On *no attempt* the value is recorded as `NULL`; may attempt another

name	first	second	third
Alice	pass	NULL	NULL
Bob	fail	pass	NULL
Carol	fail	fail	pass
Dave	fail	NULL	fail
Eve	fail	fail	NULL

> **Question**
>
> For each student, find the result of their last attempted quiz.

Common SQL Constructs

Conditional Expressions

COALESCE Expression

name	first	second	third
Alice	pass	NULL	NULL
Bob	fail	pass	NULL
Carol	fail	fail	pass
Dave	fail	NULL	fail
Eve	fail	fail	NULL

name	result
Alice	pass
Bob	pass
Carol	pass
Dave	fail
Eve	fail

Scenario

COALESCE (price, 0)

A student may attempt a quiz *up to* 3 times until they pass. Each potential attempt is recorded as follows:

- On *passing* the value is recorded as `'pass'`; cannot attempt another
- On *failing* the value is recorded as `'fail'`; may attempt another
- On *no attempt* the value is recorded as `NULL`; may attempt another

> **Question**
>
> For each student, find the result of their last attempted quiz.

'fail'

```
SELECT name,
       COALESCE (third, second, first) AS result -- the order matter!
FROM   Quiz;
```

Common SQL Constructs

Aggregation
Grouping
HAVING
Conditional
- CASE
- COALESCE
- NULLIF

Conditional Expressions

NULLIF Expression

Syntax

```
NULLIF(<value_1>, <value_2>)
```

Notes:

- Returns `NULL` if `<value_1> = <value_2>`, otherwise returns `<value_1>`

Examples

```
SELECT
  NULLIF(1,1)
  AS val;
```

val
NULL

```
SELECT
  NULLIF(2,1)
  AS val;
```

val
2

```
SELECT
  NULLIF(NULL,1)
  AS val;
```

val
NULL

```
SELECT
  NULLIF(2,NULL)
  AS val;
```

val
2

Common SQL Constructs

Aggregation
Grouping
HAVING
Conditional
- CASE
- COALESCE
- NULLIF

Conditional Expressions

NULLIF Expression

> **Question**
>
> Find the minimum and average price of pizza across different restaurants. However, since *unknown* values are represented as 0 *(instead of `NULL`)*, *exclude the unknown values from the computation*.

Incorrect

```
SELECT MIN(price) AS min_price,
       AVG(price) AS avg_price
FROM   Sells;
```

min_price	avg_price
0	18.5555555555555556

Correct

```
SELECT MIN(NULLIF(price,0)) AS min_price,
       AVG(NULLIF(price,0)) AS avg_price
FROM   Sells;
```

min_price	avg_price
16	20.8750000000000000

Structuring Queries

Structuring Queries

Table Expressions

Motivational Example

> **Question**
>
> Find the 3 cheapest pizza ***BUT*** ordered in descending order of price?

Incomplete Solution

```sql
SELECT *
FROM   Sells
WHERE  price IN (
    SELECT price FROM Sells ORDER BY price ASC LIMIT 3
  )
ORDER BY price DESC;
```

- The problem is we are only looking at the price which may contains duplicate
  - Even worse, limiting this to 3 does not even give us the 2 cheapest!

Table "Sells"

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	19
Mammas Place	Marinara	19
Pizza King	Diavola	17
Pizza King	Hawaiian	19

Structuring Queries

Table Expressions

Motivational Example

> **Question**
>
> Find the 3 cheapest pizza ***BUT*** ordered in descending order of price?

Complete Solution

```
SELECT *
FROM   (
    SELECT * FROM Sells
    ORDER BY price ASC LIMIT 3
  ) T1
ORDER BY price DESC;
```

- This is like a *"temporary"* table
  (of course, no actual tables are actually created)
- This is not yet *"common"* table expression because it can only be used by outer query

Table "Sells"

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	19
Mammas Place	Marinara	19
Pizza King	Diavola	17
Pizza King	Hawaiian	19

Structuring Queries

Table Expressions

Common Computation

Question

For each pizza with Cheese as one of its ingredient, find the restaurant names that sells it and the customer names that likes it such that the customer and restaurant are in the same area.

Common Table Expressions

Compute the highlighted part as a *"temporary"* table.

Then we can use this *"temporary"* table as if it is a table.

```
WITH
  PizzaWithCheese (pizza) AS (
    SELECT DISTINCT pizza FROM Recipes
    WHERE ingredient = 'Cheese'
  )
SELECT S.pizza, R.rname, C.cname
FROM   Sells S, Likes L, Restaurants R, Customers C
WHERE  S.rname = R.rname AND C.cname = L.cname
  AND  S.pizza = L.pizza AND R.area = C.area
  AND  S.pizza IN ( SELECT * FROM PizzaWithCheese );
```

Structuring Queries

Table Expressions

Important Notes

You are only allowed ***up to 2*** CTEs *for each question* for assignment and midterm.

Common Table Expression Syntax

Syntax

```
WITH
  CTE_1 AS ( Q_1 ),
  CTE_2 AS ( Q_2 ),
    ⋮,
  CTE_n AS ( Q_n )
Q_0
-- main SELECT statement
```

Usage:

- Each `CTE_i` is the ***name*** of a temporary table defined by query `Q_i`
- Each `CTE_i` can reference any other `CTE_j` that has been declared before `CTE_i` *(i.e., $j < i$)*
- The main `SELECT` statement `Q_0` can reference *any* possible subset of all `CTE_i`
  - It does not have to reference all, but if there are any `CTE_i` not referenced by any, it can be deleted

Note:

- The goal of using CTEs is *not* to write less code
- CTEs help to improve readability, debugging, and maintenance
- The resulting column names can also be specified as `CTE_i (<attr>, <attr>, ..., <attr>) AS ( Q_i )`

Structuring Queries

Table Expressions

Extended Example

> **Question**
>
> For each pizza with Cheese as one of its ingredient, find the restaurant names that sells it and the customer names that likes it such that the customer and restaurant are in the same area.

```
WITH
  PizzaWithCheese (pizza) AS (
    SELECT DISTINCT pizza FROM Recipes
    WHERE ingredient = 'Cheese'
  ),
  RestaurantWithCheese (rname, pizza, area) AS (
    SELECT DISTINCT rname, pizza, area
    FROM   Sells NATURAL JOIN Restaurants
    WHERE  pizza IN ( SELECT * FROM PizzaWithCheese )
  ),
  CustomerLikesCheese (cname, pizza, area) AS (
    SELECT DISTINCT cname, pizza, area
    FROM   Likes NATURAL JOIN Customers
    WHERE  pizza IN ( SELECT * FROM PizzaWithCheese )
  )
SELECT pizza, rname, cname
FROM   RestaurantWithCheese
       NATURAL JOIN
       CustomerLikesCheese;
```

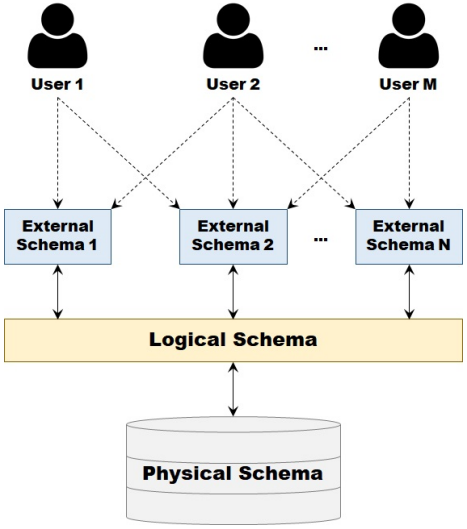
Structuring Queries

VIEWS

Virtual Tables

Common Observations

- Often only parts of a table *(i.e., rows/columns)* are of interest
- Often not all parts of a table *(i.e., rows/columns)* should be accessible to all users *(e.g., PDPA, GDPR, etc)*
- Often the same queries and/or subqueries are regularly and frequently used



Structuring Queries

VIEWS

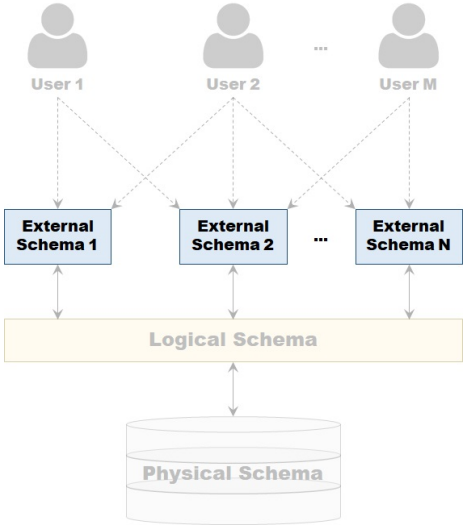
Virtual Tables

Common Observations

- Often only parts of a table *(i.e., rows/columns)* are of interest
- Often not all parts of a table *(i.e., rows/columns)* should be accessible to all users *(e.g., PDPA, GDPR, etc)*
- Often the same queries and/or subqueries are regularly and frequently used

Note

Logical schemas are tables we have created using SQL DDL *(i.e., `CREATE TABLE`)*. External schema can be represented using `VIEWS`.



Structuring Queries

VIEWS

CREATE VIEW

Syntax

```
CREATE VIEW <name> AS
  <query> -- SELECT statement
;
```

Notes:

- A `VIEW` is a *permanently* named query *(also known as virtual table)*
  - Conceptually, **NO** actual table is created
  - The computation *may* be done each time the virtual table is accessed
    (i.e., the result is conceptually not permanently stored)
- A `VIEW` can be used *almost* like a normal table
  - Some restrictions apply *(to be discussed later)*
- The resulting column names can also be specified as
  `CREATE VIEW <name> (<attr>, <attr>, ..., <attr>) AS ...`

Important Notes

You must wrap the solution of each question with the respective `VIEW` for both assignment and midterm.

Structuring Queries

VIEWS

Previous Example

> **Question**
>
> For each **pizza with Cheese** as one of its ingredient, find the restaurant names that sells it and the customer names that likes it such that the customer and restaurant are in the same area.

- Assume finding **pizza with Cheese** is a very frequent query

```
CREATE VIEW PizzaWithCheese (pizza) AS
  SELECT DISTINCT pizza FROM Recipes
  WHERE ingredient = 'Cheese';

WITH
  RestaurantWithCheese (rname, pizza, area) AS (
    SELECT DISTINCT rname, pizza, area
    FROM   Sells NATURAL JOIN Restaurants
    WHERE  pizza IN ( SELECT * FROM PizzaWithCheese )
  ),
  CustomerLikesCheese (cname, pizza, area) AS (
    SELECT DISTINCT cname, pizza, area
    FROM   Likes NATURAL JOIN Customers
    WHERE  pizza IN ( SELECT * FROM PizzaWithCheese )
  )
SELECT pizza, rname, cname
FROM   RestaurantWithCheese
       NATURAL JOIN
       CustomerLikesCheese;
```

Structuring Queries

VIEWS

Usability

SELECT Statements

- No restrictions when used in SQL queries *(i.e., `SELECT` statement)*

Updatable View

- Restrictions for `INSERT`, `UPDATE`, and/or `DELETE` statements
- Requirements:
  A. Only one entry in `FROM` clause *(table or updatable view)*
  B. No `WITH`, `DISTINCT`, `GROUP BY`, `HAVING`, `LIMIT`, or `OFFSET`
  C. No `UNION`, `INTERSECT`, `EXCEPT` or the `ALL` variants
  D. No *aggregate* functions
  E. *etc*

[#]More information can be found at https://www.postgresql.org/docs/14/sql-createview.html

Extended Concepts

Extended Concepts

Universal Quantification

Motivational Example

> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

Problems

- SQL queries is evaluated for each row
- SQL conditions in `WHERE` clause is checked row-by-row
  - But this is a property that may need to be satisfied by different rows

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mammas Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

cname	pizza
Homer	Hawaiian
Homer	Margherita
⋮	⋮

Notes

SQL directly supports only *existential* quanfitication *(`EXISTS`)*

Extended Concepts

Universal
- *Motivation*
- ***Transformation***
- *Double Negation*
- *Cardinality*

Recursive

Universal Quantification

Query Transformation

> restaurant that sells *all* pizzas liked by 'Homer'

⇒

> there *does not exist* pizza that 'Homer' likes and *not sold* by the restaurant

Note

In other words *(a logical approximation of why, can be captured as)*
∀x : Exist(x) ⇔ ¬¬∀x : Exist(x) ⇔ ¬∃x : ¬Exist(x)

Useful Query

> all pizzas that 'Homer' likes and restaurant with rname = *R* does not sell

```sql
SELECT L.pizza
FROM   Likes L
WHERE  L.cname = 'Homer'
  AND  NOT EXISTS (
         SELECT 1
         FROM   Sells S
         WHERE  S.pizza = L.pizza
           AND  S.rname = 'R'
       );
```

Try It Out!

Replace R with
- Corleone Corner
- Gambino Oven
- Lorenzo Tavern
- Mammas Place
- Pizza King

Extended Concepts

Universal Quantification

Motivational Example

> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

Double Negation Solution

```sql
SELECT DISTINCT S1.rname
FROM   Sells S1
WHERE  NOT EXISTS (
    SELECT 1 FROM Likes L
    WHERE  L.cname = 'Homer'
      AND  NOT EXISTS (
            SELECT 1 FROM Sells S
            WHERE  S.pizza = L.pizza
              AND  S.rname = S1.rname
          )
  );
```

> We need to replace `'R'` with all possible restaurants, so we put another query outside and connect it to the innermost subquery *(i.e., `S.rname = S1.rname`)*.

[#]While not overly common, SQL queries requiring universal quantification can get "ugly".

Extended Concepts

Universal
- Motivation
- Transformation
- Double Negation
- Cardinality
Recursive

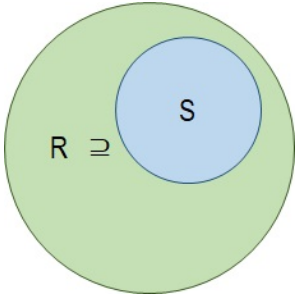
Universal Quantification

Cardinality Solution

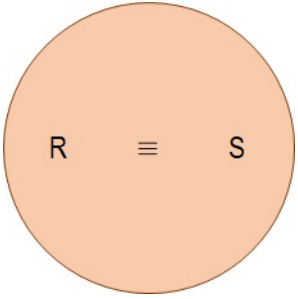
> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

- Let *R* be the set of all pizzas sold by a restaurants *(e.g., `'Corleone Corner'`)*
- Let *L* be the set of all pizzas liked by a customer *(e.g., `'Homer'`)*

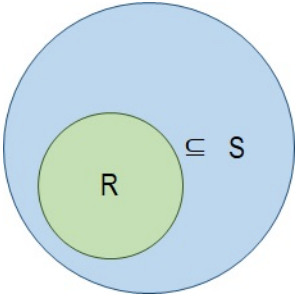
Superset



Equal



Subset



Extended Concepts

Universal Quantification

Cardinality Solution

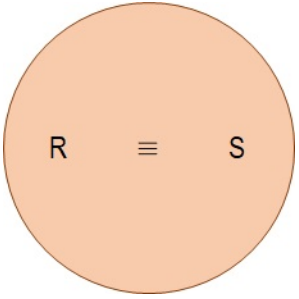
> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

- Let *R* be the set of all pizzas sold by a restaurants *(e.g., `'Corleone Corner'`)*
- Let *S* be the set of all pizzas liked by a customer *(e.g., `'Homer'`)*

Superset

- $R \supseteq S$
  $\leftrightarrow R \cup S = R$
  $\leftrightarrow |R \cup S| = |R|$

Equal



Subset

- $R \subseteq S$
  $\leftrightarrow R \cap S = R$
  $\leftrightarrow |R \cap S| = |R|$

Extended Concepts

Universal Quantification

Cardinality Solution

> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

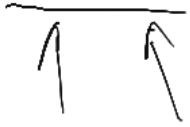
- Let *R* be the set of all pizzas sold by a restaurants *(e.g., `'Corleone Corner'`)*
- Let *L* be the set of all pizzas liked by a customer *(e.g., `'Homer'`)*

Superset

- $R \supseteq S$

  $\leftrightarrow R \cup S = R$

  $\leftrightarrow |R \cup S| = |R|$



Equal

- $R \equiv S$

  $\leftrightarrow R \supseteq S \wedge R \subseteq S$

  $\leftrightarrow |R \cup S| = |R \cap S|$

Subset

- $R \subseteq S$

  $\leftrightarrow R \cap S = R$

  $\leftrightarrow |R \cap S| = |R|$

Extended Concepts

Universal Quantification

Cardinality Solution

> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

```sql
SELECT DISTINCT rname
FROM   Sells S
WHERE  (
    SELECT COUNT(DISTINCT pizza)
    FROM (
      SELECT pizza FROM Sells S1 WHERE S1.rname = S.rname
      UNION
      SELECT pizza FROM Likes WHERE cname = 'Homer'
    ) AS T1
  ) = (
    SELECT COUNT(DISTINCT pizza)
    FROM   Sells S1 WHERE S1.rname = S.rname
  );
```

Extended Concepts

Universal
- Motivation
- Transformation
- Double Negation
- Cardinality
Recursive

Universal Quantification

Cardinality Solution

> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

```sql
SELECT DISTINCT rname
FROM   Sells S
WHERE  (
    SELECT COUNT(DISTINCT pizza)
    FROM (
      SELECT pizza FROM Sells S1 WHERE S1.rname = S.rname
      UNION
      SELECT pizza FROM Likes WHERE cname = 'Homer'
    ) AS T1
  ) = (
    SELECT COUNT(DISTINCT pizza)
    FROM   Sells S1 WHERE S1.rname = S.rname
  );
```

Note

R

```sql
SELECT pizza
FROM   Sells S1
WHERE  S1.rname
       =
       S.rname
```

Extended Concepts

Universal
- Motivation
- Transformation
- Double Negation
- Cardinality
Recursive

Universal Quantification

Cardinality Solution

> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

```sql
SELECT DISTINCT rname
FROM   Sells S
WHERE  (
    SELECT COUNT(DISTINCT pizza)
    FROM (
      SELECT pizza FROM Sells S1 WHERE S1.rname = S.rname
      UNION
      SELECT pizza FROM Likes WHERE cname = 'Homer'
    ) AS T1
  ) = (
    SELECT COUNT(DISTINCT pizza)
    FROM   Sells S1 WHERE S1.rname = S.rname
  );
```

Note

S

```sql
SELECT pizza
FROM   Likes
WHERE  cname
       =
       'Homer'
```

Extended Concepts

Universal Quantification

Cardinality Solution

> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

```sql
SELECT DISTINCT rname
FROM   Sells S
WHERE  (
    SELECT COUNT(DISTINCT pizza)
    FROM (
      SELECT pizza FROM Sells S1 WHERE S1.rname = S.rname
      UNION
      SELECT pizza FROM Likes WHERE cname = 'Homer'
    ) AS T1
  ) = (
    SELECT COUNT(DISTINCT pizza)
    FROM   Sells S1 WHERE S1.rname = S.rname
  );
```

> **Note**
>
> $R \cup S$

Extended Concepts

Universal
- Motivation
- Transformation
- Double Negation
- Cardinality
Recursive

Universal Quantification

Cardinality Solution

> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

```sql
SELECT DISTINCT rname
FROM   Sells S
WHERE  (
    SELECT COUNT(DISTINCT pizza)
    FROM (
      SELECT pizza FROM Sells S1 WHERE S1.rname = S.rname
      UNION
      SELECT pizza FROM Likes WHERE cname = 'Homer'
    ) AS T1
  ) = (
    SELECT COUNT(DISTINCT pizza)
    FROM   Sells S1 WHERE S1.rname = S.rname
  );
```

Note

$|R \cup S|$

Renaming to `T1` is needed for subquery

Extended Concepts

Universal
- Motivation
- Transformation
- Double Negation
- Cardinality
Recursive

Universal Quantification

Cardinality Solution

> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

```sql
SELECT DISTINCT rname
FROM   Sells S
WHERE  (
    SELECT COUNT(DISTINCT pizza)
    FROM (
      SELECT pizza FROM Sells S1 WHERE S1.rname = S.rname
      UNION
      SELECT pizza FROM Likes WHERE cname = 'Homer'
    ) AS T1
  ) = (
    SELECT COUNT(DISTINCT pizza)
    FROM   Sells S1 WHERE S1.rname = S.rname
  );
```

Note

$|R|$

Extended Concepts

Universal
- Motivation
- Transformation
- Double Negation
- Cardinality
Recursive

Universal Quantification

Cardinality Solution

> **Question**
>
> Find the name of a restaurant that sells *all* pizzas liked by 'Homer'.

```sql
SELECT DISTINCT rname
FROM   Sells S
WHERE  (
    SELECT COUNT(DISTINCT pizza)
    FROM (
      SELECT pizza FROM Sells S1 WHERE S1.rname = S.rname
      UNION
      SELECT pizza FROM Likes WHERE cname = 'Homer'
    ) AS T1
  ) =
    SELECT COUNT(DISTINCT pizza)
    FROM   Sells S1 WHERE S1.rname = S.rname
);
```

Scalar

Scalar

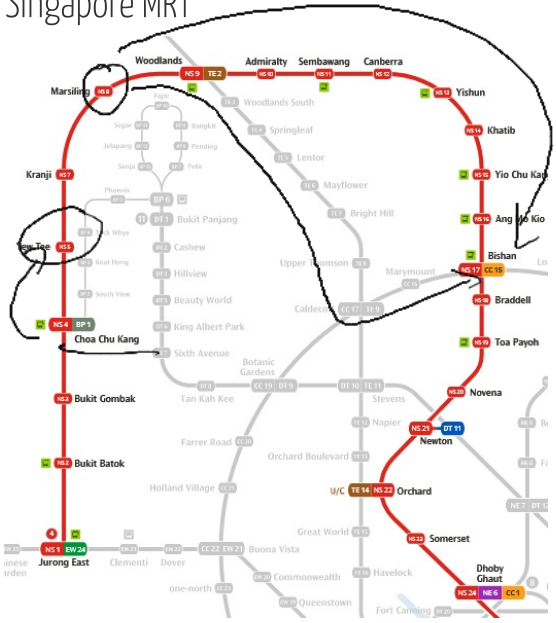
> **Note**
>
> $|R \cup S| = |R|$

Extended Concepts

Universal
Recursive
- MRT
- Query
- Recursive CTE
- Lazy Evaluation

Recursive Queries

Singapore MRT



Schema

```
CREATE TABLE MRT (
  fr_stn CHAR(5),
  to_stn CHAR(5),
  PRIMARY KEY (fr_stn, to_stn)
);
```

Sample

fr_stn	to_stn
NS4	NS5
NS5	NS7
EW5	EW6
EW6	EW7

Extended Concepts

Recursive Queries

Interesting Query

> **Question**
>
> Find all MRT station that can be reached from NS1 in 1-stop.

Potential Solution

```sql
SELECT DISTINCT to_stn
FROM   MRT


WHERE  fr_stn = 'NS1';
```

Table

to_stn
NS2

1 row

Extended Concepts

Universal
Recursive
- MRT
- Query
- Recursive CTE
- Lazy Evaluation

Recursive Queries

Interesting Query

> **Question**
>
> Find all MRT station that can be reached from NS1 in 2-stops.

Potential Solution

```
SELECT DISTINCT M2.to_stn
FROM   MRT M1,
       MRT M2

WHERE  M1.fr_stn = 'NS1'
  AND  M1.to_stn = M2.fr_stn;
```

Table

to_stn
NS3

1 row

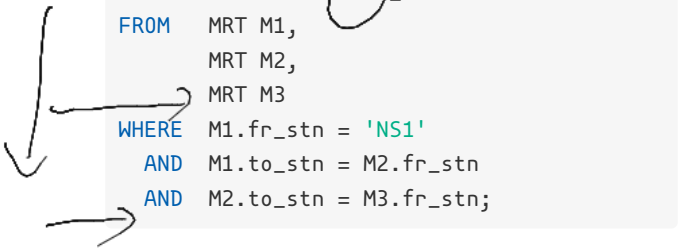
Extended Concepts

Recursive Queries

Interesting Query

> **Question**
>
> Find all MRT station that can be reached from NS1 in 3-stops.

Potential Solution



```sql
SELECT DISTINCT M3.to_stn
FROM   MRT M1,
       MRT M2,
       MRT M3
WHERE  M1.fr_stn = 'NS1'
  AND  M1.to_stn = M2.fr_stn
  AND  M2.to_stn = M3.fr_stn;
```

Table

to_stn
NS4

1 row

> **Note**
>
> What about *n*-stops? What is the maximum *n*?

Extended Concepts

Universal
Recursive
- MRT
- Query
- Recursive CTE
- Lazy Evaluation

Recursive Queries

Recursive CTE

Syntax

```
WITH RECURSIVE
  CTE_name AS (
    Q_1
    UNION [ ALL ]
    Q_2 ( CTE_name )
  )
Q_0 ( CTE_name )
```

Notes:

- `Q_1` is non-recursive
- `Q_2` is recursive and can reference `CTE_name`
- Query is evaluated "lazily", stops when a *fixed-point* is reached
  (i.e., no additional rows need to be printed)

Extended Concepts

Universal
Recursive
- MRT
- Query
- Recursive CTE
- Lazy Evaluation

Recursive Queries

Recursive CTE

> **Question**
>
> Find all MRT station that can be reached from NS1 in *at most* 3-stops.

Recursive Solution

```
WITH RECURSIVE
  Linker(to_stn, stops) AS (
    SELECT to_stn, 0
    FROM   MRT
    WHERE  fr_stn = 'NS1'
    UNION ALL
    SELECT M.to_stn, L.stops + 1
    FROM   Linker L, MRT M
    WHERE  L.to_stn = M.fr_stn
      AND  L.stops < 2
  )
SELECT DISTINCT (to_stn)
FROM   Linker;
```

Table

to_stn
NS2
NS3
NS4

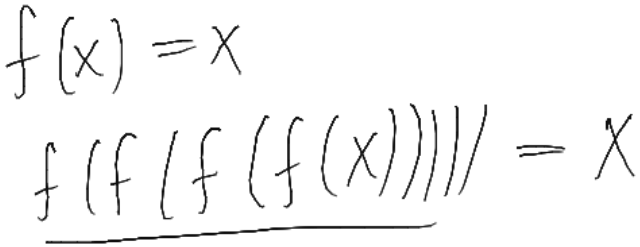
3 rows

Extended Concepts

Universal
Recursive
- *MRT*
- *Query*
- *Recursive CTE*
- ***Lazy Evaluation***

Recursive Queries

Lazy Evaluation



> **Question**
>
> Find all MRT station that can be reached from NS1 in *at most* 3-stops.

Recursive Solution

```sql
WITH RECURSIVE
  Linker(to_stn, stops) AS (
    SELECT to_stn, 0
    FROM   MRT
    WHERE  fr_stn = 'NS1'
    UNION ALL
    SELECT M.to_stn, L.stops + 1
    FROM   Linker L, MRT M
    WHERE  L.to_stn = M.fr_stn
  )
SELECT DISTINCT (to_stn)
FROM   Linker
WHERE  stops < 3;
```

Table

to_stn
NS2
NS3
NS4

3 rows

Summary

Summary

What We Have Covered

SQL Query

- Most common vocabulary for writing queries
- Basic means to "organize" complex queries (CTEs, Views)
- Extended features
  - Universal quantification
  - Recursive queries

Advantages

- More maintainable SQL query
- Easily extendable query

Summary

Covered
Limitations

Limitations of SQL Queries

Poorly Supported Queries

- "*Sorted by pizza price, are there somewhere in the ranking 3 pizzas that are sold by the same restaurant listed in a row*"
- Queries/tasks common for time-series: moving average, sliding window, *etc*

Common Approaches

- Keep or move logic into the application
- Use features that make SQL "*Turing-complete*"* *(e.g., stored procedures))*
- Use different data model / DBMS *(e.g., graph database for recursive queries, or use time-series databases)*

*Strictly speaking, the support of recursive CTEs already made SQL "Turing-complete".

```
postgres=# exit
Press any key to continue . . .
```

Solutions

Quiz #1

> **Quiz #1**
>
> What is the problem with the invalid query?

Hint

> Look at the table created from the query on the right. Try to do you own grouping using `rname` as the grouping criteria. What do you see? Can you answer what is the value of `pizza` for all group?

```
SELECT rname, pizza, SUM(price)
FROM   Sells;
```

Solutions

Quiz #2

> **Quiz #2**
>
> What is the "problem" with the *valid* query?

Comment

> Since `iso2` is the `PRIMARY KEY` of the table `Countries`, every single row can be uniquely identified by their `iso2` column. In other words, each "group" is a group of a single row.
>
> Which brings us to the weird behaviour by PostgreSQL on why it does not allow candidate key *(i.e., `UNIQUE` constraint, even with `NOT NULL`)* to be used as a grouping.

Solutions

Quiz #3

> **Quiz #3**
>
> Will it be valid if we do `SELECT R.rname, C.area ...` instead of `SELECT R.rname, R.area ...`?

Comment

> Test it out and see if there is an error. This is a simplified case of the rule. By `R.area = C.area`, we know that if `R.area` is unique in the result, so should `C.area`. Is PostgreSQL looking into this "fact" before they are running the code?

Solutions

Quiz #4

> **Quiz #4**
>
> Will it be valid if we do `SELECT R.rname, C.area ...` instead of `SELECT R.rname, R.area ...`?

Comment

> Given that in the `GROUP BY` clause, `rname` and `pizza` is the primary key of `Sells`, each row is in a group of its own. Therefore, the condition `price > 20` can be applied to each row. Can you not use `HAVING` clause in this case? If so, what will be the value of `COUNT(*)`?

Solutions

HAVING Clause

Exercise #1

> **Question**
>
> Find all restaurants that sells any pizza cheaper than any pizza sold by Lorenzo Tavern.

Query

Result

rname
Pizza King
Mammas Place
Gambino Oven
Corleone Corner