CS2102: Database Systems

SQL (Part 2)

Clarifications

Clarifications

Candidate Keys
Weak Equivalence

Candidate Keys

NULL or Not NULL?

Consideration

- Consider *R(A,B,C,D)*
  - Let the primary key be *{A}*
  - Let *{B,C}* be a candidate key

A	B	C	D
1	NULL	1	0
2	NULL	1	1

line 1
line 2

{B, C}

Problems?

- Does *{A}* uniquely identify *{D}*?
- Does *{B,C}* uniquely identify *{D}*?

Notes

This problem arise because postgresql does not consider *(**NULL**, 1)* to be *different* from *(**NULL**, 1)* using =.

In postgre sql, can be inserted after line 2. Therefore, candidate key {B, C} does not uniquely identify D.

Clarifications

Candidate Keys
Weak Equivalence

Weak Equivalence

Mathematical Example

- Consider the sequence $1 + q + q^2 + q^3 + \ldots + q^n$ for $q \leq 1$
- Is it equivalent to $(1 - q^n) / (1 - q)$?
  - If $q \neq 1$ then yes
  - If $q = 1$ then $(1 - q^n) / (1 - q)$ is *undefined* *(i.e., error)*

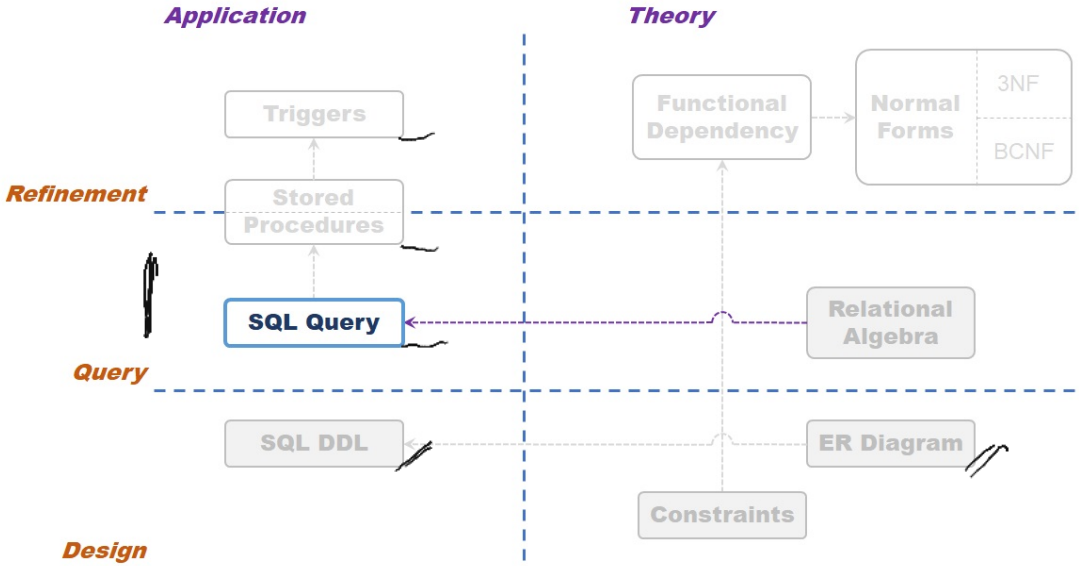
Relational Algebra

- Note that using our construct, relational algebra expression is either *always* error *(i.e., invalid expression)* or *never* error
  - Still useful for generic expressions where some elements are left unspecified *(e.g., $\sigma_{[C]}(R - S)$ is only weakly equivalent to $\sigma_{[C]}(R) - S$)*

SQL

- SQL query has a construct that is only checked at run-time *(e.g., scalar subquery)* which may or may not produce an error depending on the input relation

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

SQL vs. Relational Algebra

Basic SQL Queries

- Simple Queries
- Set Operations
- Join Queries

Composing

- Subqueries

Ordering

- Ordering
- Ranking

SQL vs. Relational Algebra

SQL vs. Relational Algebra

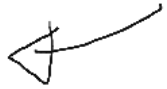
Comparison

Comparison Table

	Relational Algebra	**SQL**
Paradigm	Imperative *(query language using operators to perform queries)*	Declarative *(query language built on top of relational algebra)*
Semantic	Set *(i.e., no duplicate rows/tuples)*	Multiset/Bag *(i.e., allows duplicate rows/tuples)*
Query	Relational algebra expression	SELECT statement

SELECT Statement

```
SELECT  [ DISTINCT ] <target-list>
FROM    <relation-list>
[ WHERE <conditions> ];
```



SQL vs. Relational Algebra

Basic Form

SELECT-FROM-WHERE

```
SELECT    [ DISTINCT ]  <target-list>      -- SELECT clause
FROM      <relation-list>                  -- FROM clause
WHERE     <conditions>                     -- WHERE clause
```

Notes

- `[ DISTINCT ]` By default, duplicates are **NOT** eliminated. Explicit specification of `DISTINCT` is required to eliminate duplicates.
- `<target-list>` List of attributes/columns of the output result.
- `<relation-list>` List of relations that are relevant to answer the given query.
- `<conditions>` Conditions that specify a valid result row/tuple

(based on principle of acceptance).

- ***Order-Independent:*** The order of rows in the result does not matter*(unsorted/unranked)*

SQL vs. Relational Algebra

Basic Form

SELECT-FROM-WHERE

```
SELECT    [ DISTINCT ]  <target-list>      -- SELECT clause
FROM      <relation-list>                  -- FROM clause
WHERE     <conditions>                     -- WHERE clause
```

Notes

- `[ DISTINCT ]` — By default, duplicates are **NOT** eliminated. Explicit specification of `DISTINCT` is required to eliminate duplicates.
- `<target-list>` — List of attributes/columns of the output result.
- `<relation-list>` — List of relations that are relevant to answer the given query.
- `<conditions>` — Conditions that specify a valid result row/tuple

(based on principle of acceptance).

- ***Order-Independent:*** The order of rows in the result does not matter*(unsorted/unranked)*

Convention

We will assume that the expected output:
- Should not contain duplicate
- Order-independent *(unless specified otherwise such as with ranking)*

[#] We will not penalize if you are using `DISTINCT` keywords everywhere as long as the final results satisfy the requirement.

SQL vs. Relational Algebra

Conceptual Evaluation

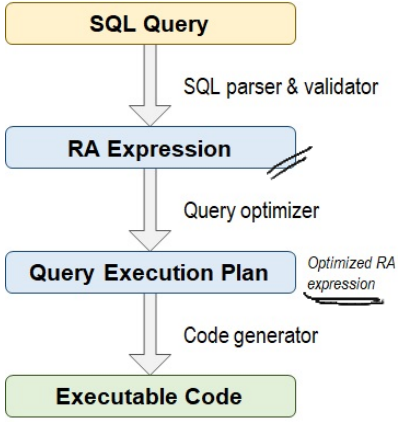
Preprocessing: SQL Query to Executable Code

SELECT Statement

```
SELECT  DISTINCT a1, a2, ..., am
FROM    r1, r2, ..., rn
WHERE   c
```

Relational Algebra

$$\pi_{[a1, a2, ..., am]}(\sigma_{[c]}(r1 \times r2 \times ... \times rn))$$



SQL vs. Relational Algebra

Conceptual Evaluation

Evaluation Steps

SELECT Statement

```
SELECT   DISTINCT a1, a2, ..., am
FROM     r1, r2, ..., rn
WHERE    c
```

1. Compute cross product of all tables *(r1, r2, ..., rn)*

Relational Algebra

$$\pi_{[a1, a2, ..., am]}(\sigma_{[c]}(r1 \times r2 \times ... \times rn))$$

SQL vs. Relational Algebra

Conceptual Evaluation

Evaluation Steps

SELECT Statement

```
SELECT   DISTINCT a1, a2, ..., am
FROM     r1, r2, ..., rn
WHERE    c
```

1. Compute cross product of all tables *(r1, r2, ..., rn)*
2. Keep only rows where condition is true *(c)*

Relational Algebra

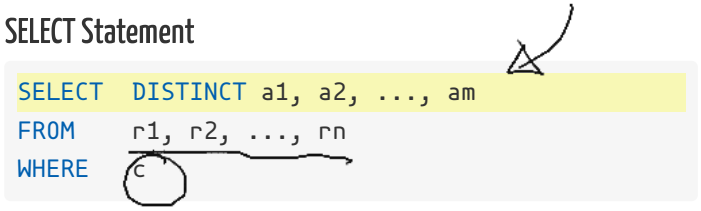
$$\pi_{[a1, a2, ..., am]}(\sigma_{[c]}(r1 \times r2 \times ... \times rn))$$

SQL vs. Relational Algebra

Conceptual Evaluation

Evaluation Steps

SELECT Statement



```
SELECT  DISTINCT a1, a2, ..., am
FROM    r1, r2, ..., rn
WHERE   c
```

1. Compute cross product of all tables *(r1, r2, ..., rn)*
2. Keep only rows where condition is true *(c)*
3. Keep only attributes specified *(a1, a2, ..., am)*

Relational Algebra

$$\pi_{[a1, a2, ..., am]}(\sigma_{[c]}(r1 \times r2 \times ... \times rn))$$

SQL vs. Relational Algebra

Conceptual Evaluation

Evaluation Steps

SELECT Statement

```
SELECT [DISTINCT] a1, a2, ..., am
FROM    r1, r2, ..., rn
WHERE   c
```

1. Compute cross product of all tables *(r1, r2, ..., rn)*
2. Keep only rows where condition is true *(c)*
3. Keep only attributes specified *(a1, a2, ..., am)*
4. Remove duplicate rows *(only if distinct is specified)*

Relational Algebra

$$\pi_{[a1, a2, ..., am]}(\sigma_{[c]}(r1 \times r2 \times ... \times rn))$$

SQL vs. Relational Algebra

Conceptual Evaluation

Evaluation Steps

SELECT Statement

```
SELECT  DISTINCT a1, a2, ..., am
FROM    r1, r2, ..., rn
WHERE   c
```

1. Compute cross product of all tables *(r1, r2, ..., rn)*
2. Keep only rows where condition is true *(c)*
3. Keep only attributes specified *(a1, a2, ..., am)*
4. Remove duplicate rows *(only if distinct is specified)*

Relational Algebra

$$\pi_{[a1, a2, ..., am]}(\sigma_{[c]}(r1 \times r2 \times ... \times rn))$$

Notes

This relational algebra mapping is a conceptual mapping without consideration for performance

SQL vs. Relational Algebra

Basic Strategy

Problem Solving

Subproblems

- Can you find a simpler problem to solve or solve a partial problem?
  - If so, solve the simpler/partial problem as subqueries
  - Then solve the main problem

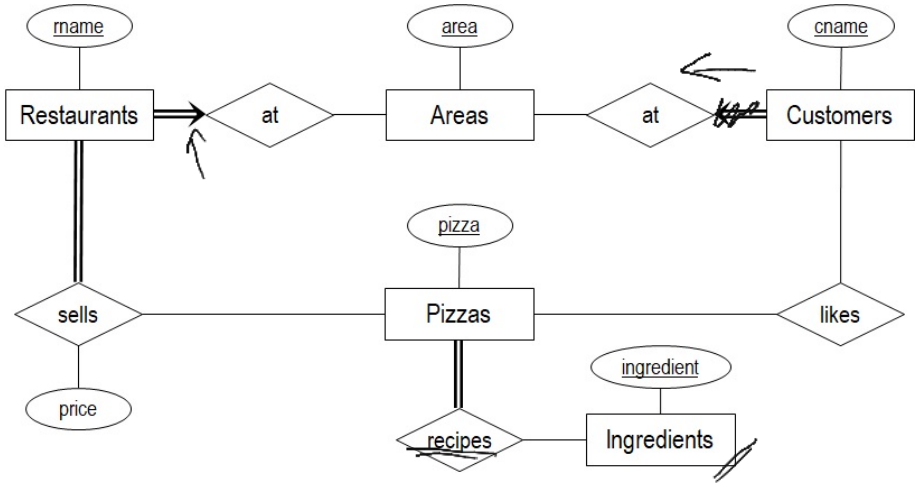
Problem Deconstruction

- Answer the following questions
  - What columns/attributes to find?
  - Which relations containing the attributes?
  - What are the conditions to be satisfied?

SQL vs. Relational Algebra

Example Database

ER Diagram



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

SQL vs. Relational Algebra

Universal Schema Assumption

Example Database

Schema

Relations

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

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 (L05.sql)

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

SQL vs. Relational Algebra

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
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

Basic SQL Queries

Basic SQL Queries

Simple Queries

Simple Condition

Question

Find the restaurants and the pizzas that they sell where the price is under $20.

Problem Deconstruction

- What columns/attributes to find?
- Which relations have the attributes?
- What are the conditions to be satisfied?

Basic SQL Queries

Simple Queries

Simple Condition

> **Question**
>
> Find the restaurants and the pizzas that they sell where the price is under $20.

Problem Deconstruction

- What columns/attributes to find?
- Which relations have the attributes?
- What are the conditions to be satisfied?

`rname`, `pizza` — SELECT
`Sells` — FROM
`price` under $20 — WHERE

price < 20

Basic SQL Queries

Simple Queries

Simple Condition

> **Question**
>
> Find the restaurants and the pizzas that they sell where the price is under $20.

Problem Deconstruction

- What columns/attributes to find? `rname`, `pizza`
- Which relations have the attributes? `Sells`
- What are the conditions to be satisfied? `price` under $20

SQL

```
SELECT rname, pizza
FROM   Sells
WHERE  price < 20;
```

Loop is applied.

rname	pizza
Corleone Corner	Margherita
Gambino Oven	Siciliana
Pizza King	Diavola

Basic SQL Queries

rname	pizza
Corleone Corner	Hawaiian
Gambino Oven	Siciliana
Lorenzo Tavern	Funghi
Mammas Place	Marinara
Pizza King	Hawaiian

Simple Queries

Complex Condition

Exercise #1

Find the resturants and the pizzas that they sell where either (i) the price is over $20 or the pizza is Siciliana and (ii) the pizza is not Diavola.

```
1  -- What columns/attributes to find?
2  -- Which relations have the attributes?
3  -- What are the conditions to be satisfied?
4
5  (price > 20 OR pizza = 'Siciliana') AND (pizza <> 'Diavola')
```

Basic SQL Queries

Simple Queries

All Attributes

> **Question**
>
> Find the restaurants, the pizzas that they sell, and the price of the pizza where the price is under $20.

Wild Card

- We can use wild card * to include all attributes

SQL

```
SELECT *
FROM   Sells
WHERE  price < 20;
```

rname	pizza	price
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Pizza King	Diavola	17

Basic SQL Queries

SELECT Clause

Expressions

Common Use Cases

- Combine and process attribute values
- Rename columns

> **Question**
>
> Find the restaurants, the pizzas that they sell, and the price of the pizza in SGD *(current exchange rate is 1 USD = 1.36 SGD)*.

SQL

```
SELECT rname, pizza,
       'S$' || (price * 1.36) AS sgd
FROM   Sells;
```

rname	pizza	sgd
Corleone Corner	Diavola	S$32.64
Corleone Corner	Hawaiian	S$34.00
⋮ Corner	⋮	⋮
Pizza King	Hawaiian	S$28.56

Basic SQL Queries

SELECT Clause

Expressions

Common Use Cases

- Combine and process attribute values
- Rename columns

Operations

- **Mathematical** `+`, `*`, `-`, `%`, `/`, `^`, `|/`, *etc*
- **String** `||` *(concatenate)*, `LOWER(s)`, `UPPER(s)`, *etc*
- **Date Time** `+`, `NOW()`, *etc*

Renaming

- `column AS alias`

> **Question**
>
> Find the restaurants, the pizzas that they sell, and the price of the pizza in SGD *(current exchange rate is 1 USD = 1.36 SGD)*.

SQL

```
SELECT rname, pizza,
       'S$' || (price * 1.36) AS sgd
FROM   Sells;
```

rname	pizza	sgd
Corleone Corner	Diavola	**S$32.64**
Corleone Corner	Hawaiian	**S$34.00**
⋮ Corner	⋮	⋮
Pizza King	Hawaiian	**S$28.56**

Basic SQL Queries

SELECT Clause

Duplicates

Multiset Semantics

- By default, SQL does **NOT** eliminate duplicates
- Keyword `DISTINCT` to enforce duplicate elimination

> **Question**
>
> Find all pizza names that is sold by at least one restaurant.

SQL

```
SELECT pizza
FROM   Sells;
```

pizza
Diavola
Hawaiian
⋮
Diavola
Hawaiian

8 rows

```
SELECT DISTINCT pizza
FROM   Sells;
```

pizza
Hawaiian
Margherita
⋮
Siciliana

6 rows

> **Notes**
>
> The order of the output when using `DISTINCT` may be different. This depends on how duplicates are removed.

Basic SQL Queries

x	y	x IS DISTINCT FROM y	x IS NOT DISTINCT FROM y
not-NULL	not-NULL	x <> y	x = y
not-NULL	NULL	True	False
NULL	not-NULL	True	False
NULL	NULL	False	True

SELECT Clause

Duplicates

NULL Values

- `DISTINCT` keyword checks for *distinct* rows using `IS DISTINCT FROM`

```
SELECT area
FROM   Customers;
```

area
West
South
East
Central
Central
Central
Central
North
NULL
NULL

```
SELECT DISTINCT area
FROM   Customers;
```

area
NULL
East
South
West
North
Central

Basic SQL Queries

WHERE Clause

Conditions for NULL Values

- Finding tuples with NULL or not-NULL as attribute values
  - `x IS NULL` vs `x = NULL`
  - `x IS NOT NULL` vs `x <> NOT NULL`

> **Question**
> Find all customer names with unknown area.

Notes

There is no *(syntax)* error when using = or <> operator. But the output may not be what you expect!

NULL = NULL
→ evaluates to NULL

NULL IS NULL
→ evaluates to True.

SQL

IS NULL

```
SELECT cname
FROM   Customers
WHERE  area IS NULL;
```

cname
Adi
Yoga

2 rows

= NULL

```
SELECT cname
FROM   Customers
WHERE  area = NULL;
```

cname

0 rows

Basic SQL Queries

WHERE Clause

Pattern Matching

numeric, alphabet, symbols

Basic Pattern Matching ([NOT] LIKE)

- `_` matches *any* single character
- `%` matches any sequence of *0 or more* characters

Advanced Patter Matching

- *Regular expression*
  (out of scope)

> **Question**
> Find all pizza that start with "Ma" and end with "a".

Maav MaX

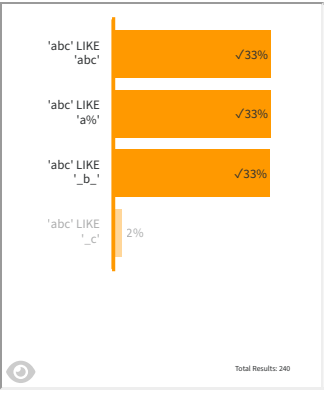
SQL

```sql
SELECT pizza
FROM   Pizzas
WHERE  pizza LIKE 'Ma%a';
```

pizza
Marinara
Margherita

> **Notes**
> Some pattern may actually be simplified. For example, `%_` can be simplified to `%`.

Basic SQL Queries



WHERE Clause

Pattern Matching

Basic Pattern Matching ([NOT] LIKE)

- `_` matches *any* single character
- `%` matches any sequence of *0 or more* characters

> **Exercise 2**
>
> Which of the following conditions below returns TRUE?

	Choice	Comment	
A	`'abc' LIKE 'abc'`		?
B	`'abc' LIKE 'a%'`		?
C	`'abc' LIKE '_b_'`		?
D	`'abc' LIKE '_c'`		?

Basic SQL Queries

Recap

Two relations *R* and *S* are **union-compatible** if it satisfies both of the following:

- *R* and *S* have the *same number of attributes*
- The corresponding attributes have the *same or compatible domains*

Set Operations

Operations

Let Q_1 and Q_2 be two SQL queries that yield ***union-compatible*** tables

- Q_1 `UNION` Q_2 = $Q_1 \cup Q_2$
- Q_1 `INTERSECT` Q_2 = $Q_1 \cap Q_2$
- Q_1 `EXCEPT` Q_2 = $Q_1 - Q_2$

Table "R"

v1
1
2
2

Table "S"

v2
2
3

Important

- Eliminate duplicate rows from result
- ~~Column names~~ follow the left relation *(i.e., Q_1)*
- The ordering of the output result may be different

UNION

v1
1
2
3

INTERSECT

v1
2

EXCEPT

v1
1

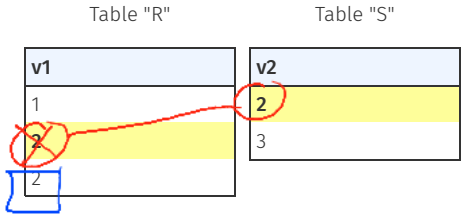
Basic SQL Queries

Set Operations

Duplication

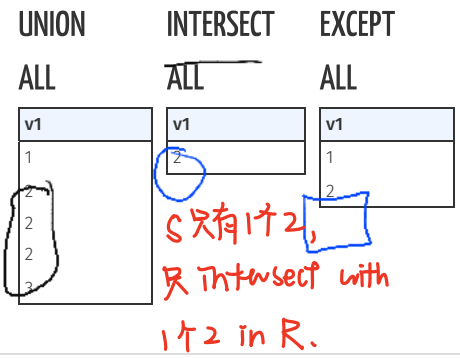
The following operations do ***NOT*** remove duplicates

- Q_1 `UNION ALL` Q_2
- Q_1 `INTERSECT ALL` Q_2
- Q_1 `EXCEPT ALL` Q_2



Important

- May produce unintuitive result *(but explainable)*
  - Basically, each element is treated as distinct element
  - Intersection keeps/removes only distinct elements



Basic SQL Queries

Set Operations

Example

Find all pizza names with both cheese and chilli as the ingredients.

Cheese Only

```
SELECT pizza
FROM   Recipes
WHERE  ingredient
       = 'Cheese';
```

pizza
Diavola
Margherita
Siciliana

Chilli Only

```
SELECT pizza
FROM   Recipes
WHERE  ingredient
       = 'Chilli';
```

pizza
Diavola
Marinara

Cheese and Chilli

```
SELECT pizza
FROM   Recipes
WHERE  ingredient
       = 'Cheese'
INTERSECT
SELECT pizza
FROM   Recipes
WHERE  ingredient
       = 'Chilli';
```

pizza
Diavola

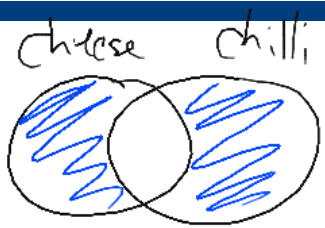
Notes

This is an example of *decomposition* of problem into smaller problems.

1. Find all ... with cheese only
2. Find all ... with chilli only

Basic SQL Queries

Set Operations



Exercise #3

Find all pizza names with *either* cheese or chilli *(or both)* as the ingredients.

```sql
-- What operations do you need?
SELECT pizza
FROM   Recipes
WHERE  ingredient
       = 'Cheese'
UNION
SELECT pizza
FROM   Recipes
WHERE  ingredient
       = 'Chilli';
```

Basic SQL Queries

Set Operations

small world assumption:
If you can't find it in the table, it must be false.

Exercise #4

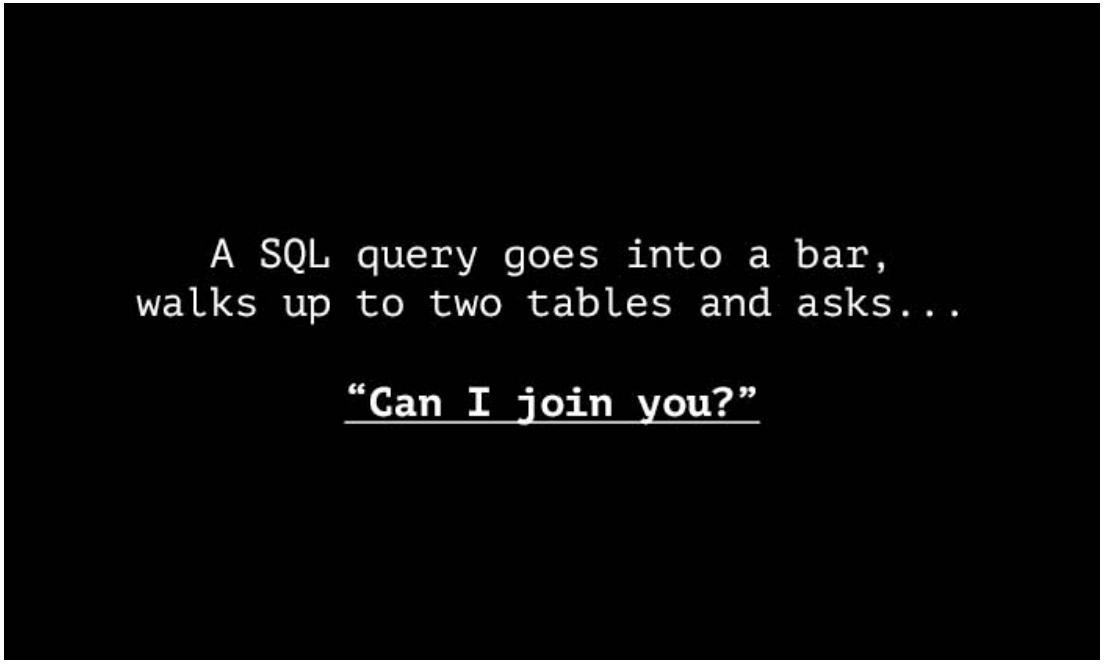
Find all customer names that does not like any pizza.

```
1  -- What operations do you need?
2
3  SELECT cname FROM Customers
4  EXCEPT
5  SELECT cname FROM Likes;
```

Basic SQL Queries

Simple Queries
SELECT Clause
WHERE Clause
Set Operations
Join Operations
- Restriction
- Multi-Relational
- Natural Join
- Outer Join

Join Operations



Basic SQL Queries

r1. rname

Join Operations

Restriction

- A table name *(after renaming)* can appear at most once in the `FROM` clause
- We can rename a table to allow the same table to appear more than once in the `FROM` clause *(but with different name)*

Example

Question

Find all pair of restaurant (R_1, R_2) that are in the same area such that $R_1 < R_2$.

Error

```
SELECT rname, rname
FROM   Restaurants, Restaurants
WHERE  rname < rname
  AND  area = area;
```

No Error

```
SELECT r1.rname, r2.rname
FROM   Restaurants AS r1, Restaurants r2
WHERE  r1.rname < r2.rname
  AND  r1.area = r2.area;
```

[#]The keyword `AS` here is *optional*.

Basic SQL Queries

Join Operations

Multi-Relational Queries

- Most common in practice: *cross product* + *selection condition* + *attribute selection*

> **Example**
>
> For all pizzas with cheese as its ingredient, find the restaurant that sells the pizza as well as the price of the pizza.

Notes

Ingredients are in the table *Recipes* while prices are in the table *Sells*. So, we need to join the table.

θ - join

Equivalent Queries

```
SELECT DISTINCT rname, price
FROM   Sells S, Recipes R
WHERE  S.pizza = R.pizza
  AND  R.ingredient = 'Cheese';
```

```
SELECT DISTINCT rname, price
FROM   Sells S JOIN Recipes R
    ON S.pizza = R.pizza
WHERE  R.ingredient = 'Cheese';
```

can be "and"

rname	price
Corleone Corner	19
Corleone Corner	24
Gambino Oven	16
Pizza King	17

[#]The operation `JOIN` is equivalent to the more explicit operation `INNER JOIN`. In other words, the use of `INNER` is *optional*.

Basic SQL Queries

Join Operations

Natural Join

- Check for equivalence of common attributes

> **Example**
>
> For all pizzas with cheese as its ingredient, find the restaurant that sells the pizza as well as the price of the pizza.

Natural Join

```
SELECT DISTINCT rname, price
FROM   Sells S NATURAL JOIN Recipes R
-- no need S.pizza = R.pizza, why?
WHERE  R.ingredient = 'Cheese';
```

rname	price
Corleone Corner	19
Corleone Corner	24
Gambino Oven	16
Pizza King	17

> This is why ***universal schema assumption*** is useful!

Basic SQL Queries

Join Operations

Outer Join

- Recall that sometimes the "*dangling tuples*" are of interest

> **Example**
>
> For all customer names that does not like any pizza.

Outer Join

```
SELECT DISTINCT C.cname
FROM   Customers C LEFT JOIN Likes L
    ON C.cname = L.cname
WHERE  L.pizza IS NULL; -- keep only dangling tuples
```

cname
Adi
Willie
Yoga

Notes

`LEFT JOIN` is equivalent to the more explicit `LEFT OUTER JOIN` *(i.e., the use of `OUTER` is optional)*. There are also `RIGHT [OUTER] JOIN`, `FULL [OUTER] JOIN` and the natural variant *(e.g., `NATURAL LEFT JOIN` without the use of `ON`)*.

[#]Of course, using `EXCEPT` is probably easier. This shows that there are multiple solutions for the same problem.

Composing

$P \longrightarrow Q \equiv \neg P \vee Q.$

Composing

Q1 UNION Q2

Subqueries / Nested Queries

Q1, Q2 are not subqueries

Preliminaries

- Subqueries are queries *within* another queries
  - Can appear in `SELECT` clause
  - Can appear in `FROM` clause *(for next week)*
  - Can appear in `WHERE` clause

- Can be used to construct complex queries
  - Subquery expressions:
    - `IN/NOT IN` subqueries
    - `EXISTS/NOT EXISTS` subqueries
    - `ANY/~~SOME~~` subqueries (`SOME` *is not the keyword in postgresql*)
    - ~~`ALL`~~ subqueries

Composing

Scalar Subqueries

Definition

A **scalar subquery** is a query that returns *at most a single value* *(i.e., a table with 1 row and 1 column).*

Note:

- This is *dynamically checked* at run-time
- Can be used as a *value* in queries
- If the result returns 0 row, then it is treated as NULL

Examples

Scalar

attr
CS2102

Scalar

attr

≡ NULL

Not Scalar

attr
1
2

Not Scalar

attr1	attr2
CS2102	A+

Composing

Scalar Subqueries

Example #1

Find all area, the pizza sold in the area, and all the prices of the pizza in that area.

Using Scalar Subquery

```
SELECT (SELECT area
        FROM   Restaurants R
        WHERE  R.rname = S.rname),
       pizza, price
FROM   Sells S;
```

area	pizza	price
North	Diavola	24
North	Hawaiian	25
North	Margherita	19
Central	Siciliana	16
Central	Funghi	23
South	Marinara	22
East	Diavola	17
East	Hawaiian	21

Note

Alternatives using joins are also possible.

Composing

Scalar Subqueries

Example #2

Find all restaurant names, pizza, and the price of the pizza that sells pizza cheaper than the price of Margherita sold by Corleone Corner.

Using Scalar Subquery

```
SELECT *
FROM   Sells
WHERE  price < (SELECT S.price
                FROM   Sells S
                WHERE  rname = 'Corleone Corner'
                  AND  pizza = 'Margherita');
```

rname	pizza	price
Gambino Oven	Siciliana	16
Pizza King	Diavola	17

Composing

[NOT] IN Subqueries

Syntax

```
   :
WHERE <expr> [ NOT ] IN [ <subquery> | <tuple> ];
```

Note:

- `<tuple>` specification is $(value_1, value_2, ..., value_n)$

Semantics

- For the query to work, the subquery **must** return exactly one column
- `IN` returns `TRUE` if `<expr>` matches *any* subquery row
  - $\exists$ *row* $\in$ *<subquery>* : *row* = *<expr>*
- `NOT IN` returns `TRUE` if `<expr>` matches *no* subquery row
  - $\forall$ *row* $\in$ *<subquery>* : *row* <> *<expr>*

[#] Remember, we have ***three-valued logic***.

Composing

[NOT] IN Subqueries

Example #1

Find all restaurant names that sells pizza with Cheese as one of its ingredients.

Outer Query

```
SELECT DISTINCT rname
FROM   Sells
WHERE  pizza IN
```

Inner Query

all pizza w/ cheese

```
(SELECT pizza
 FROM   Recipes
 WHERE  ingredient = 'Cheese');
```

Result

rname
Corleone Corner
Gambino Oven
Pizza King

Quiz #1

Which `pizza` is being referred to in the *inner query*? Is it (A) `Sells.pizza` or (B) `Recipes.pizza`?

Composing

[NOT] IN Subqueries

Example #2

Find all customer names and area such that the customer do not like any pizza.

Outer Query

```
SELECT cname, area
FROM   Customers
WHERE  cname NOT IN
```

Inner Query

```
(SELECT cname
 FROM   Likes);
```

Result

cname	area
Willie	North
Adi	NULL
Yoga	NULL

Rule of Thumb

- `IN` subqueries can typically be replaced with inner joins
- `NOT IN` subqueries can typically be replaced with outer joins

Composing

Row Constructor

[NOT] IN Subqueries

Example #3

Find all pizza and price sold by either Gambino Oven or Pizza King.

Explicit Tuple

```
SELECT pizza, price
FROM   Sells
WHERE  rname IN ('Gambino Oven', 'Pizza King');
```

Result

pizza	price
Siciliana	16
Diavola	17
Hawaiian	21

Composing

ANY/SOME Subqueries

Syntax

```
   :
WHERE <expr> <op> ANY <subquery>;
```

Note:

- `<op>` are relational operators *(e.g., =, <>, <, <=, >, >=)*
- `SOME` is used in some other DBMS *(PostgreSQL uses `ANY`)*

Semantics

- For the query to work, the subquery **must** return exactly one column
- The expression `<expr>` is compared to each row from subquery using the operator `<op>`
- `ANY` returns `TRUE` if comparison evaluates to `TRUE` for *at least one* row in the subquery
  - ∃ *row* ∈ *subquery* : (*<expr> <op> row*) = TRUE

[#]Remember, we have ***three-valued logic***.

Composing

ANY/SOME Subqueries

Example

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

(also possible, sells any pizza cheaper than the most expensive pizza sold by Lorenzo Tavern).

Outer Query

```
SELECT DISTINCT rname
FROM   Sells
WHERE  price <
```

Inner Query

```
ANY (SELECT price
     FROM   Sells
     WHERE  rname = 'Lorenzo Tavern');
```

Result

rname
Pizza King
Mammas Place
Gambino Oven
Corleone Corner

Composing

ALL Subqueries

Syntax

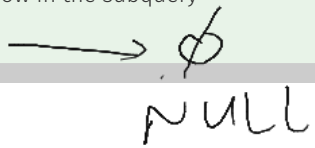
```
   :
WHERE <expr> <op> ALL <subquery>;
```

Note:

- `<op>` are relational operators *(e.g., =, <>, <, <=, >, >=)*

Semantics

- For the query to work, the subquery **must** return exactly one column
- The expression `<expr>` is compared to each row from subquery using the operator `<op>`
- `ALL` returns `TRUE` if comparison evaluates to `TRUE` for *all* row in the subquery
  - ∀ *row* ∈ *subquery* : (*<expr> <op> row*) = TRUE



[#] Remember, we have ***three-valued logic***.

Composing

Subqueries
Scalar
[NOT] IN
ANY/SOME
ALL
- Syntax/Semantics
- Examples
[NOT] EXISTS
Scoping
Nested
Remarks

ALL Subqueries

> **Example**
>
> Find all restaurants that sells any pizza more expensive than all pizzas sold by Mammas Place *(also possible, sells any pizza more expensive than the most expensive pizza sold by ...)*.

Outer Query

```
SELECT DISTINCT rname
FROM   Sells
WHERE  price >
```

Inner Query

```
ALL (SELECT price
     FROM   Sells
     WHERE  rname = 'Mammas Place');
```

Result

rname
Lorenzo Tavern
Corleone Corner

Composing

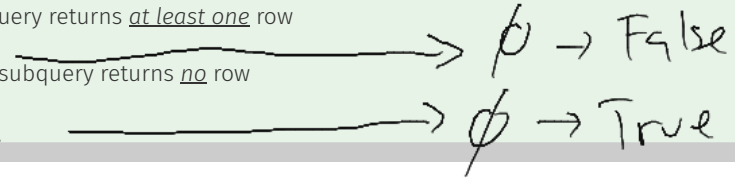
[NOT] EXISTS Subqueries

Syntax

```
  :
WHERE [ NOT ] EXISTS <subquery>;
```

Semantics

- The subquery may return any number of columns
- `EXISTS` returns `TRUE` if the subquery returns *at least one* row
  - ∃ *row* ∈ *subquery* : TRUE
- `NOT EXISTS` returns `TRUE` if the subquery returns *no* row
  - ∀ *row* ∈ *subquery* : FALSE



[#] You read that right, there is no operation within the "forall" *(i.e., ∀)* and "there exists" *(i.e., ∃)*

Composing

[NOT] EXISTS Subqueries

> **Example #1**
>
> Find all areas for which Diavola is sold by at least one resturant.

Outer Query

```
SELECT DISTINCT area
FROM   Restaurants R
WHERE
```

Inner Query

```
EXISTS (SELECT 1 -- ???
        FROM   Sells S
        WHERE  S.rname = R.rname
          AND  S.pizza = 'Diavola');
```

Result

area
East
North

> **Quiz #2**
>
> Why do we use `SELECT 1`?

Composing

[NOT] EXISTS Subqueries

Example #1

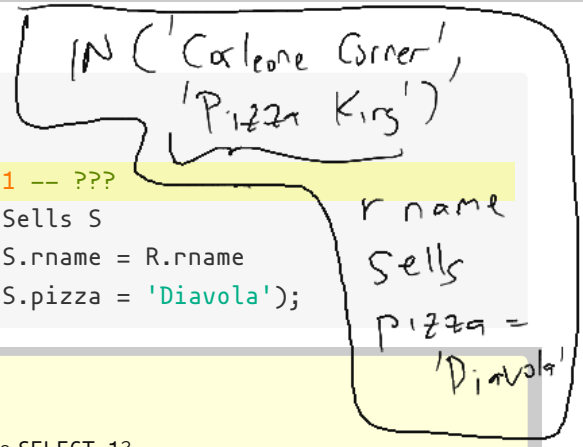
Find all areas for which Diavola is sold by at least one restaurant

Outer Query

```
SELECT DISTINCT area
FROM   Restaurants R
WHERE
```

Inner Query

```
EXISTS (SELECT 1 -- ???
        FROM   Sells S
        WHERE  S.rname = R.rname
          AND  S.pizza = 'Diavola');
```



Result

area
East
North

Quiz #2

Why do we use `SELECT 1`?

Notes

Let's fix `R.rname` to Pizza King. What is the result of the inner query?

```
SELECT 1 FROM Sells S
WHERE    S.rname = 'Pizza King'
  AND    S.pizza = 'Diavola';
```

Composing

[NOT] EXISTS Subqueries

Example #2

Find all customer names and areas for which there is no restaurant in the area.

Outer Query

```
SELECT *
FROM   Customers C
WHERE
```

Inner Query

```
NOT EXISTS (SELECT 1
            FROM   Restaurants R
            WHERE  C.area = R.area);
```

Result

cname	area
Homer	West
Adi	NULL
Yoga	NULL

Quiz #3

Why is `NULL` part of the result? Can we remove this and if so, how? *(**Hint:** it's not `IS NOT DISTINCT FROM`)*

Composing

Scoping

Correlated Subqueries

- Inner query uses value from outer query
  - Uncorrelated subqueries for `[ NOT ] EXISTS` are usually either wrong or unnecessary
- Result of inner query *typically* depends on the value of the outer query
  - This may result in potentially slow performance

Naming Ambiguities

- Same attribute names in both inner and outer queries *(e.g., `price`)*
- Resolve ambiguities using table aliases *(e.g., `S1` and `S2`)*
  - Otherwise, scoping rules applies

S1.price

S2.price

```
SELECT DISTINCT S1.rname
FROM   Sells S1
WHERE  price < ANY
    (SELECT price
     FROM   Sells S2
     WHERE  S2.rname = 'Lorenzo Tavern');
```

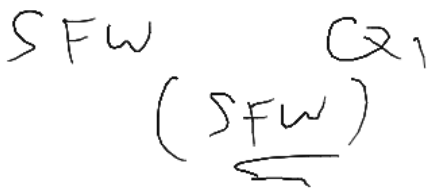
Composing

Scoping

Scoping Rules

- A table alias declared in a *(sub-)*query Q can only be used in Q or subqueries nested within Q
- If the same table alias is declared in both subquery Q_1 and in outer query Q_0 *(or not at all)*, the declaration in Q_1 is applied

General Rule:
- From inner to outer queries in case of multiple nestings



Composing

Scoping

Scoping Rules

- A table alias declared in a *(sub-)*query Q can only be used in Q or subqueries nested within Q
- If the same table alias is declared in both subquery Q_1 and in outer query Q_0 *(or not at all)*, the declaration in Q_1 is applied

General Rule:
- From inner to outer queries in case of multiple nestings

Examples

```
SELECT DISTINCT S1.rname
FROM   Sells S1
WHERE  price < ANY
    (SELECT price
     FROM   Sells S2
     WHERE  S2.rname = 'Lorenzo Tavern');
```

Notes

Table aliases are used
- `S1.rname` from `S1`
- `S2.rname` from `S2`

Composing

Scoping

Scoping Rules

- A table alias declared in a *(sub-)*query Q can only be used in Q or subqueries nested within Q
- If the same table alias is declared in both subquery Q_1 and in outer query Q_0 *(or not at all)*, the declaration in Q_1 is applied

General Rule:
- From inner to outer queries in case of multiple nestings

Examples

```
SELECT DISTINCT S1.rname
FROM   Sells S1
WHERE  price < ANY -- S1.price
    (SELECT price  -- S2.price
     FROM   Sells S2
     WHERE  S2.rname = 'Lorenzo Tavern');
```

Notes

Table aliases are **NOT** used and should refer to the "closest" scope *(from inside to outside)*

Composing

Scoping

Possible Confusion

> **Example #1**
>
> For each restaurant, find the *most expensive* pizza *(if there are multiple, return all)*.

Correct

```
SELECT *
FROM   Sells S1
WHERE  S1.price >= ALL
  (SELECT S2.price
   FROM   Sells S2
   WHERE  S2.rname = S1.rname);
```

rname	pizza	price
Corleone Corner	Hawaiian	25
Gambino Oven	Siciliana	16
⋮	⋮	⋮

Incorrect

```
SELECT *
FROM   Sells S1
WHERE  S1.price >= ALL
  (SELECT S2.price
   FROM   Sells S2
   WHERE  S2.rname = rname);
```

rname	pizza	price
Corleone Corner	Hawaiian	25

> Where are the rest?

Composing

Scoping

Possible Confusion

> **Example #1**
>
> For each restaurant, find the *most expensive* pizza *(if there are multiple, return all)*.

Correct

```
SELECT *
FROM   Sells S1
WHERE  S1.price >= ALL
  (SELECT S2.price
   FROM   Sells S2
   WHERE  S2.rname = S1.rname);
```

rname	pizza	price
Corleone Corner	Hawaiian	25
Gambino Oven	Siciliana	16
⋮	⋮	⋮

Incorrect

```
SELECT *
FROM   Sells S1
WHERE  S1.price >= ALL
  (SELECT S2.price
   FROM   Sells S2
   WHERE  S2.rname = rname);
```

rname	pizza	price
Corleone Corner	Hawaiian	25

> Where are the rest?

Where Are The Rest?

`S2.rname = rname` is basically `S2.rname = S2.rname` if we follow the scoping rules. But `S2.rname = S2.rname` is always true for non-`NULL` values.

Composing

Scoping

Possible Mistakes

> **Example #2**
>
> Find all customer names with a known area located in an area *without* a restaurant.

Good

```
SELECT T1.cname
FROM   Customers T1
WHERE  T1.area NOT IN
  (SELECT T2.area
   FROM   Restaurants T2);
```

cname
Homer

Okay-ish *(can be confusing)*

```
SELECT T.cname
FROM   Customers T
WHERE  T.area NOT IN
  (SELECT T.area
   FROM   Restaurants T);
```

cname
Homer

Wrong

```
SELECT T1.cname
FROM   Customers T1
WHERE  T1.area NOT IN
  (SELECT T1.area
   FROM   Restaurants T2);
```

cname

Composing

Scoping

Possible Mistakes

> **Example #2**
>
> Find all customer names with a known area located in an area *without* a restaurant.

Good

```
SELECT T1.cname
FROM   Customers T1
WHERE  T1.area NOT IN
  (SELECT T2.area
   FROM   Restaurants T2);
```

cname
Homer

Okay-ish *(can be confusing)*

```
SELECT T.cname
FROM   Customers T
WHERE  T.area NOT IN
  (SELECT T.area
   FROM   Restaurants T);
```

cname
Homer

Wrong

```
SELECT T1.cname
FROM   Customers T1
WHERE  T1.area NOT IN
  (SELECT T1.area
   FROM   Restaurants T2);
```

cname

> **Quiz #4**
>
> What happened to the wrong code?

Composing

Multiple Nested Subqueries

> **Example**
>
> Find all customers in Central area that likes at least one pizza sold by Corleone Corner *(exclude customers that do not like any pizza)*.

Outer Query

```
-- Customers in central
SELECT C.cname
FROM   Customers C
WHERE  area = 'Central'
  AND  C.cname IN
```

Inner Query #1

```
-- Likes at least one pizza
(SELECT L.cname
 FROM   Likes L
 WHERE  L.pizza IN


);
```

Inner Query #2

```
-- Sold by Corleone Corner
(SELECT S.pizza
 FROM   Sells S
 WHERE  S.rname
        = 'Corleone Corner')
```

Result

cname
Ralph

Composing

Remarks

Required Constructs

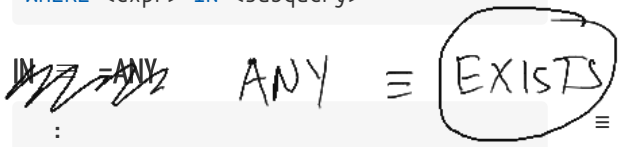
- Not all constructs are *absolutely* required

IN ≡ =ANY

```
  :
WHERE <expr> IN <subquery>
```

≡

```
  :
WHERE <expr> = ANY <subquery>
```



≡

```
  :
WHERE <e1> <op> ANY (
  SELECT <e2>
  FROM   <rel>
  WHERE  <cond>
)
```

```
  :
WHERE EXISTS(
  SELECT *
  FROM   <rel>
  WHERE  <cond> AND <e1> <op> <e2>
)
```

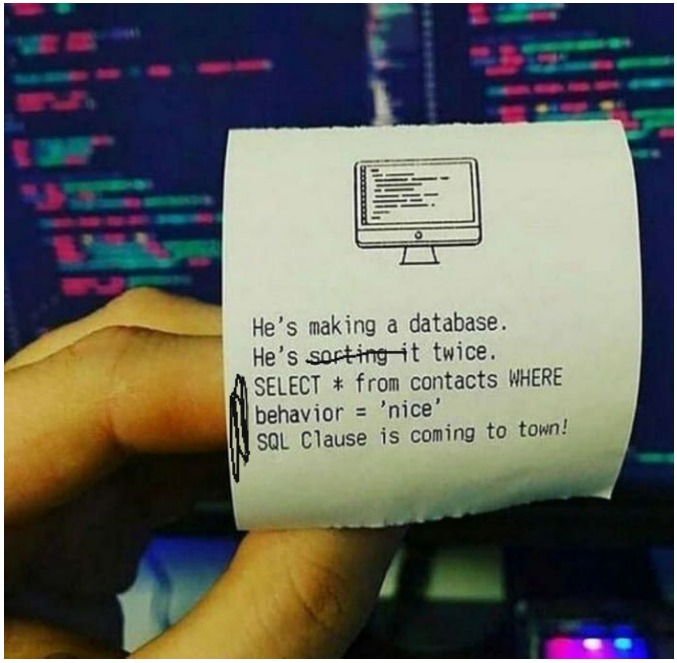
Ordering

Ordering

ORDER BY
- *Preliminary*
- *Multi-Attribute*
- *Usage*

LIMIT & OFFSET

ORDER BY



Ordering

ORDER BY

Preliminary

- By default, order of rows in a table is left up to the DBMS
- Sorting of rows by a single attribute can be done by `ORDER BY`
  - `ORDER BY <attribute> ASC` : Ascending Order *(default, `ASC` can be omitted)*
  - `ORDER BY <attribute> DESC`: Descending Order

- If duplicate removal is needed *(e.g., `DISTINCT`)*, the attribute being sorted ***must*** appear in `SELECT` clause
- Sorting with respect to multiple attributes and different ordering are supported

> **Example**
>
> Find all restaurants and their pizzas and prices sorted in descending order of price.

SQL

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

Ordering

ORDER BY

Multi-Attribute Sorting

> **Example**
>
> Find all pizza with recipes containing Cheese sorted in descending order of pizza and for each pizza sort in ascending order of ingredient.

SQL

```
SELECT *
FROM   Recipes
WHERE  pizza IN
  (SELECT pizza
   FROM   Recipes
   WHERE  ingredient = 'Cheese')
ORDER BY pizza DESC,
         ingredient ASC;
```

Result

pizza	ingredient
Siciliana	Anchovies
Siciliana	Caspers
Siciliana	Cheese
Margherita	Cheese
Margherita	Tomato
Diavola	Cheese
Diavola	Chilli
Diavola	Salami

Ordering

ORDER BY

Real-World Usage

Ranking

Pos	Club	P	W	D	L	GD	Pts
1 ↑	Liverpool	28	21	6	1	49	69
2 ↓	Manchester City	28	22	2	4	55	68
3 -	Tottenham Hotspur	28	20	0	8	26	60
4 ↑	Arsenal	28	17	5	6	22	56
5 ↓	Manchester United	28	16	7	5	19	55
6 -	Chelsea	27	16	5	6	18	53
7 -	Wolverhampton Wanderers	28	11	7	10	0	40
8 -	Watford	28	11	7	10	-1	40
9 -	Everton	28	10	6	12	0	36
10 -	West Ham United	28	10	6	12	-6	36

Ordering

LIMIT & OFFSET

Ranking

- Returning only a portion of the result table
  - `LIMIT k` Return the "*first*"* `k` rows of the result table
  - `OFFSET i` Specify the position of the "*first*" row to be considered
- `LIMIT` and `OFFSET` are typically meaningful *only in combination* with `ORDER BY`

> **Example**
>
> Find the 3 cheapest pizza, the restaurant selling them, and the price.

```
SELECT *
FROM   Sells
ORDER BY price ASC
LIMIT 3;
```

Result

rname	pizza	price
Gambino Oven	Siciliana	16
Pizza King	Diavola	17
Corleone Corner	Margherita	19

*If there are multiple "*first*" `k` *(e.g., same ranking)*, then the order within this ordering is undetermined.

Ordering

LIMIT & OFFSET

Offset

> **Example**
>
> Find the "next" 3 cheapest pizza, the restaurant selling them, and the price.

```
SELECT *
FROM   Sells
ORDER BY price ASC
OFFSET 3
LIMIT  3;
```

Result

rname	pizza	price
Pizza King	Hawaiian	21
Mammas Place	Marinara	22
Lorenzo Tavern	Funghi	23

> **Quiz #5**
>
> Can you find the 3 cheapest pizza ***BUT*** ordered in descending order of price?

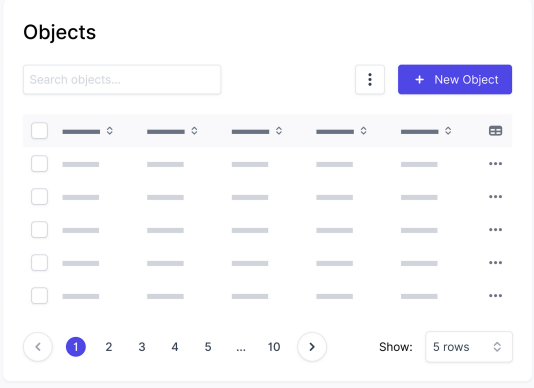
Ordering

ORDER BY
LIMIT & OFFSET
- Ranking
- Offset
- Usage

LIMIT & OFFSET

Real-World Usage

Pagination



Summary

Summary

Summary

Aggregates

Common Table Expression

Query

- Declaration query language
  - Describe ***what*** the output looks like and **NOT** ***how*** to compute the output
- Build on top of Relational Algebra

Universal Quant

Topics

- Basic — `SELECT ... FROM ... WHERE`
- Set — `UNION`, `INTERSECT`, `EXCEPT` *(and the `ALL` variants)*
- Join — `[INNER] JOIN`, `[LEFT | RIGHT | FULL] [OUTER] JOIN` *(and `NATURAL`)*
- Subqueries — `[NOT] IN`, `ANY`, `ALL`, `[NOT] EXISTS`
- Ordering — `ORDER BY`, `LIMIT`, `OFFSET`

Notes

Basic, set, and join queries have a *direct mapping* to RA expression *(apart from set vs multiset)*.

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

Solutions

Simple Queries

Complex Condition

Exercise #1

Find the resturants and the pizzas that they sell where either (i) the price is over $20 or the pizza is Siciliana and (ii) the pizza is not Diavola.

rname	pizza
Corleone Corner	Hawaiian
Gambino Oven	Siciliana
Lorenzo Tavern	Funghi
Mammas Place	Marinara
Pizza King	Hawaiian

Solutions

WHERE Clause

Pattern Matching

Basic Pattern Matching ([NOT] LIKE)

- `_` matches *any* single character
- `%` matches any sequence of *0 or more* characters

> **Exercise 2**
>
> Which of the following conditions below returns TRUE?

	Choice	Comment	
A	`'abc' LIKE 'abc'`	*YES: exact match!*	✓
B	`'abc' LIKE 'a%'`	*YES: but can also accept* `'abcd'`	✓
C	`'abc' LIKE '_b_'`	*YES: 3 characters where middle is b*	✓
D	`'abc' LIKE '_c'`	*NO: this is only two characters*	✗

Solutions

Set Operations

Exercise #3

Find all pizza names with *either* cheese or chilli *(or both)* as the ingredients.

Solutions

Set Operations

Exercise #4

Find all customer names that does not like any pizza.

Solutions

Quiz #1

> **Quiz #1**
>
> Which `pizza` is being referred to in the *inner query*? Is it (A) `Sells.pizza` or (B) `Recipes.pizza`?

Outer Scope

```
SELECT DISTINCT rname
FROM   Sells
WHERE  pizza IN
  ( SELECT pizza
    FROM   Recipes
    WHERE  ingredient = 'Cheese' );
```

> The outer query refers to `Sells.pizza`

Inner Scope

```
SELECT DISTINCT rname
FROM   Sells
WHERE  pizza IN
  ( SELECT pizza
    FROM   Recipes
    WHERE  ingredient = 'Cheese' );
```

> The inner query refers to `Recipes.pizza`

Solutions

Quiz #2

> **Quiz #2**
>
> Why do we use `SELECT 1`?

Solution

> `EXISTS` and `NOT EXISTS` only check if the table has any row or not. It does not matter what the content of the table is. Try the following:
>
> ```
> SELECT 1 FROM Restaurants;
> ```
>
> What do you see?

Solutions

Quiz #3

Quiz #3

Why is `NULL` part of the result? Can we remove this and if so, how? *(**Hint:** it's not `IS NOT DISTINCT FROM`)*

Solution

As stated, we cannot simply replace `=` with `IS NOT DISTINCT FROM`. So the following does not work.

```
SELECT * FROM Customers C
WHERE  NOT EXISTS (SELECT 1 FROM Restaurants R WHERE C.area = R.area);
```

Instead, we should add another condition to ensure that when `C.area` is `NULL`, it is excluded from the result *(since we use `NOT EXISTS`, we should put `OR C.area IS NULL`)*.

```
SELECT * FROM Customers C
WHERE  NOT EXISTS (SELECT 1 FROM Restaurants R WHERE C.area = R.area
                                                  OR  C.area IS NULL);
```

Solutions

Quiz #4

> **Quiz #4**
>
> What happened to the wrong code?

Solution

The code is reproduced on the right. Since we use `T1.area` in the *inner query*, it refers directly to the `area` of `Customers`. Notice also that at Line 3, we check if `T1.area NOT IN ( ... )`. We can then see that `x IN (x)` is always true and `x NOT IN (x)` is always false.

Because `x NOT IN (x)` is always false, then the `WHERE` clause always evaluate to false. This means, the `WHERE` clause will not match anything and the result will be empty.

```
SELECT T1.cname
FROM   Customers T1
WHERE  T1.area NOT IN
  (SELECT T1.area
   FROM   Restaurants T2);
```

Solutions

Quiz #5

Quiz #5

Can you find the 3 cheapest pizza ***BUT*** ordered in descending order of price?

Solution

One way to solve this is really to just first find all the 3 cheapest pizza as a subquery. Then we check if the pizza is part of this solution *(using `IN` or `EXISTS`)*. We can now sort this outer query in a different ordering from the inner query!

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

Note that this is an incomplete solution and breaks down if there are duplicate price. A complete solution will be shown next week when we put the subquery in the `FROM` clause.