Supplemental Material

- What are you installing when you install PostgreSQL?
  - Server-side program: the database management system itself
  - Client-side program: the client tools to manipulate the server via networks
- Search for the keywords after the class:
  - basics in computer networking (IP address, port, client, server, web browser, HTTP)
  - client-server architecture, browser-server architecture

Principles of Database Systems (CS307)
Lecture 3: Retrieving Data from One Table

Yuxin Ma

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.

Select

- `select * from [tablename]`
  - The `select` clause lists the attributes desired in the result of a query
  - To display the full content of a table, you can use select *
    - * : all columns

```
select A1, A2, ..., An
 from r1, r2, ..., rm
      where P
```

Select

- `select * from [tablename]`
  - The `select` clause lists the attributes desired in the result of a query
  - To display the full content of a table, you can use select *
    - * : all columns

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P
```

- Such a query is frequently used in interactive tools (especially when you don't remember column names ...)
  - But you should not use it, though, in application programs

Restrictions

- When tables contains thousands or millions or billions of rows, you are usually interested in only a small subset, and only want to return some of the rows

peopleid	first_name	surname	born	died

Restrictions

- Filtering
  - Performed in the "where" clause
  - Conditions are usually expressed by a column name
    - ... followed by a comparison operator and the value to which the content of the column is compared
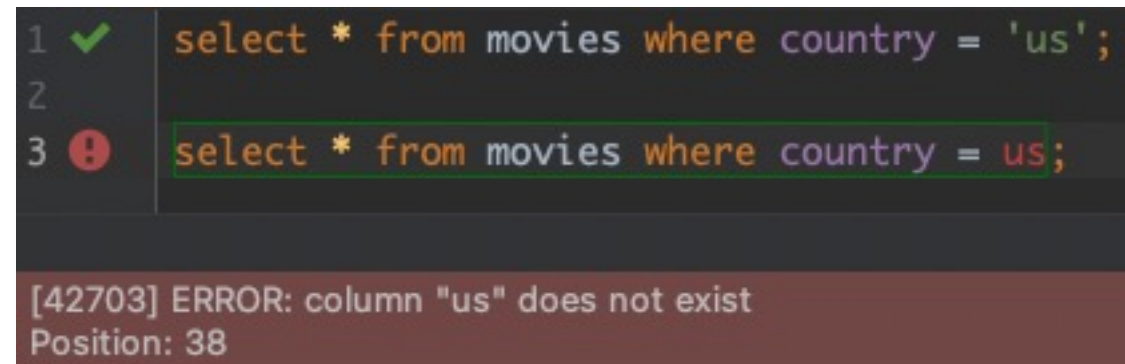 - Only rows for which the condition is true will be returned

```
select * from movies where country = 'us';
```

Comparison

- You can compare to:
  - a number
  - a string constant
  - another column (from the same table or another, we'll see queries involving several tables later)
  - the result of a function (we'll see them soon)

String Constants

- Be aware that string constants must be quoted between single-quotes
  - If they aren't quoted, they will be interpreted as column names
  - * Same thing with Oracle if they are double-quoted

```
1 ✔   select * from movies where country = 'us';
2
3 ❗   select * from movies where country = us;


[42703] ERROR: column "us" does not exist
Position: 38
```

Filtering

- Note that a filtering condition returns a subset
  - If you return all the columns from a table without duplicates, it won't contain duplicates either and will be a valid "relation"

```
select country from movies;
```

	country
1	ru
2	eg
3	ma
4	ar
5	in
6	in
7	pk
8	dk
9	jp
10	eg
11	us
12	ca
13	ru
14	be
15	br
16	my
17	cn
18	de

Select without From or Where

- An attribute can be a literal with no from clause

```
select '437'
```

  - Results is a table with one column and a single row with value "437"
  - Can give the column a name using:

```
select '437' as FOO
```

- An attribute can be a literal with from clause

```
select 'A' from movies
```

  - Result is a table with one column and N rows (number of tuples in the movies table), each row with value "A"

Select without From or Where

- An attribute can be a literal with no from clause

```
select '437'
```

A common way to test expressions

  - Results is a table with one column and a single row with value "437"
  - Can give the column a name using:

```
select '437' as FOO
```

- An attribute can be a literal with from clause

```
select 'A' from movies
```

  - Result is a table with one column and N rows (number of tuples in the movies table), each row with value "A"

Arithmetic Expression

- The `select` clause can contain arithmetic expressions involving the operation, +, −, *, and /, and operating on constants or attributes of tuples

	🔢 runtime ⬍
1	161
2	102
3	90
4	94
5	130
6	159
7	\<null\>
8	102
9	108
10	\<null\>
11	106
12	\<null\>
13	100
14	95
15	\<null\>

```
select runtime from movies
-- <--

select runtime * 10 as runtime10 from movies;  -->
```

	🔢 runtime10 ⬍
1	1610
2	1020
3	900
4	940
5	1300
6	1590
7	\<null\>
8	1020
9	1080
10	\<null\>
11	1060
12	\<null\>
13	1000
14	950
15	\<null\>

Arithmetic Expression

- The `select` clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples

	runtime
1	161
2	102
3	90
4	94
5	130
6	159
7	<null>
8	102
9	108
10	<null>
11	106
12	<null>
13	100
14	95
15	<null>

```
select runtime from movies
-- <--

select runtime * 10 as runtime10 from movies;   -->
```

as clause:
- Rename the column

	runtime10
1	1610
2	1020
3	900
4	940
5	1300
6	1590
7	<null>
8	1020
9	1080
10	<null>
11	1060
12	<null>
13	1000
14	950
15	<null>

Logical Connectives

- and, or, not
  - Just like in programming languages
  - All logical operators have different precedence
    - For example, and is "stronger" than or

Table 1-1. Operator Precedence (decreasing)

Operator/Element	Associativity	Description
::	left	PostgreSQL-style typecast
[ ]	left	array element selection
.	left	table/column name separator
-	right	unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
IS		test for TRUE, FALSE, UNKNOWN, NULL
ISNULL		test for NULL
NOTNULL		test for NOT NULL
(any other)	left	all other native and user-defined operators
IN		set membership
BETWEEN		containment
OVERLAPS		time interval overlap
LIKE ILIKE		string pattern matching
< >		less than, greater than
=	right	equality, assignment
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

https://www.postgresql.org/docs/7.2/sql-precedence.html

Logical Connectives

- and, or, not
  - Just like in programming languages
  - All logical operators have different precedence
    - For example, and is "stronger" than or.

```
select * from movies
where (country = 'us' or country = 'gb') and (year_released between 1940 and 1949);
```

Differences?

```
select * from movies
where country = 'us' or country = 'gb' and year_released between 1940 and 1949;
```

Logical Connectives

- Use parentheses to specify that the "or" should be evaluated before the "and", and that the conditions filter
  - 1) British or American films
  - 2) that were released in the 1940s

```
select * from movies
where (country = 'us' or country = 'gb') and (year_released between 1940 and 1949);
```

0 error(s), 0 warning(s)

```
select * from movies
where country = 'us' or country = 'gb' and year_released between 1940 and 1949;
```

Logical Connectives

- Question:
  - Find the Chinese movies from the 1940s and American movies from the 1950s

Logical Connectives

- Question:
  - Find the Chinese movies from the 1940s and American movies from the 1950s

```
select * from movies
where (country = 'cn'
   and year_released between 1940 and 1949)
or (country = 'us'
   and year_released between 1950 and 1959)
```

In this case, parentheses are optional – but they don't hurt
- The parentheses make the statement easier to understand

Logical Connectives

- The operands of the logical connectives can be expressions involving the comparison operators `<`, `<=`, `>`, `>=`, `=`, and `<>`.
  - Note that there are two ways to write "not equal to": `!=` and `<>`
  - Comparisons can be applied to results of arithmetic expressions
- Beware that "bigger" and "smaller" have a meaning that depends on the data type
  - It can be tricky because most products implicitly convert one of the sides in a comparison between values of differing types

```
2 < 10        -- true
'2' < '10'    -- false


'2-JUN-1883'>'1-DEC-2056'   -- single-quoted, treated as strings but not dates
```

Logical Connectives

- `in()`
    - It can be used as the equivalent for a series of equalities with `or`
    - It may make a comparison clearer than a parenthesized expression
    - * Some advanced features of `in()` will be introduced when learning subqueries

```
where (country = 'us' or country = 'gb')
  and year_released between 1940 and 1949

where country in ('us', 'gb')
  and year_released between 1940 and 1949
```

Logical Connectives

- Negation
  - All comparisons can be negated with not

```sql
-- exclude all movies selected in the previous page

where not ((country in ('us', 'gb')) and (year_released between 1940 and 1949))
where (country not in ('us', 'gb')) or (year_released not between 1940 and 1949)  -- equivalent query
```

between Comparison Operator

- between … and …
  - shorthand for: >= and <=

```sql
year_released between 1940 and 1949

-- It's shorthand for this:
year_released >= 1940 and year_released <= 1949
```

between Comparison Operator

- between … and …
  - shorthand for: >= and <=

```
year_released between 1940 and 1949

-- It's shorthand for this:
year_released >= 1940 and year_released <= 1949
```

not "<"

like

- For strings, you also have like `which` is a kind of regex (regular expression) for dummies.
- `like` compares a string to a pattern that can contain two wildcard characters:
  - `%` meaning "any number of characters, including none"
  - `_` meaning "one and only one character"

like

```
select * from movies where title not like '%A%' and title not like '%a%';

select * from movies where upper(title) not like '%A%';
-- not recommended due to the performance cost of upper()
```

- This expression for instance returns films the title of which doesn't contain any A
  - This A might be the first or last character as well
  - Note that if the DBMS is case sensitive, you need to cater both for upper and lower case
  - Function calls could slow down queries; use with caution

Date

- Date formats
  - Beware also of date formats, and of conflicting European/American formats which can be ambiguous for some dates. Common problem in multinational companies.

    DD/MM/YYYY

    MM/DD/YYYY

    YYYY/MM/DD

Date

```
select * from forum_posts where post_date >= '2018-03-12';
select * from forum_posts where post_date >= date('2018-03-12');
select * from forum_posts where post_date >= date('12 March, 2018');
```

- Whenever you are comparing data of slightly different types, you should use functions that "cast" data types
  - It will avoid bad surprises
  - The functions don't always bear the same names but exist with all products
- Default formats vary by product, and can often be changed at the DBMS level
  - So, better to use explicit date types and functions other than strings
  - Conversely, you can format something that is internally stored as a date and turn it into a character string that has almost any format you want

Date and Datetime

- If you compare datetime values to a date (without any time component) the SQL engine will not understand that the date part of the datetime should be equal to that date
  - Rather, it will consider that the date that you have supplied is actually a datetime, with the time component that you can read below
    - `date('2020-03-20')` is equal to `datetime('2020-03-20 00:00:00')`
- Date functions
  - Many useful date functions when manipulating date and datetime values
  - However, most of them are DBMS-dependent

```
select date_eq_timestamp(date('2018-03-12'), date('2018-02-12') + interval '1 month');  -- true
```

NULL

- In a language such as Java, you can compare a reference to `null`, because `null` is defined as the '0' address.
  - In C, you can also compare a pointer to `NULL` (pointer is C-speak for reference)

NULL

- Not in SQL, where NULL denotes that <u>a value is missing</u>
  - Null in SQL is <u>not</u> a value
    - … and if it's not a value, hard to say if a condition is true.
    - A lot of people talk about "null values", but they have it wrong
  - Most expression with NULL is evaluated to NULL

```
select * from movies where runtime is null;

select * from movies where runtime = null;  -- warning in DataGrip; not the same as "is null"
```

Some Functions

- Show DDL of a table

```
desc movies;   -- Oracle, MySQL

describe table movies   -- IBM DB2

\d movies   -- PostgreSQL

.schema movies   -- SQLite
```

Some Functions – Compute and Derive

- One important feature of SQL is that you don't need to return data exactly as it was stored
  - Operators, and many (*mostly DBMS specific*) functions allow to return transformed data

Some Functions

- A simple transformation is concatenating two strings together
  - Most products use || (two vertical bars) to indicate string concatenation
  - SQL Server, though, uses +, and MySQL a special `concat()` function that also exists in some other products

```sql
select title
       || ' was released in '
       || year_released movie_release
from movies
where country = 'us';
```

```
▥ movie_release
1   Variety was released in 2008
2   Inglourious Basterds was released in 2009
3   La grande vadrouille was released in 1966
4   Pulp Fiction was released in 1994
5   Schindler's List was released in 1993
6   Star Wars was released in 1977
7   The Dark Knight was released in 2008
8   The Godfather was released in 1972
9   The Shawshank Redemption was released in 1994
10  Titanic was released in 1997
11  Charade was released in 1963
12  North by Northwest was released in 1959
13  Singin' in the Rain was released in 1952
14  Rear Window was released in 1954
15  City Lights was released in 1931
```

Some Functions

- A simple transformation is concatenating two strings together
  - Most products use || (two vertical bars) to indicate string concatenation
  - SQL Server, though, uses +, and MySQL a special `concat()` function that also exists in some other products

```sql
select title
       || ' was released in '
       || year_released movie_release
from movies
where country = 'us';
```

```
▐▦ movie_release
1    Variety was released in 2008
2    Inglourious Basterds was released in 2009
3    La grande vadrouille was released in 1966
4    Pulp Fiction was released in 1994
5    Schindler's List was released in 1993
6    Star Wars was released in 1977
7    The Dark Knight was released in 2008
8    The Godfather was released in 1972
9    The Shawshank Redemption was released in 1994
10   Titanic was released in 1997
11   Charade was released in 1963
12   North by Northwest was released in 1959
13   Singin' in the Rain was released in 1952
14   Rear Window was released in 1954
15   City Lights was released in 1931
```

Note that you can give a name to an expression
- This will be used as column header
- It also becomes a "virtual column" if you turn the query into a "virtual table"

Some Functions

- A simple transformation is concatenating two strings together
  - Most products use || (two vertical bars) to indicate string concatenation
  - SQL Server, though, uses +, and MySQL a special `concat()` function that also exists in some other products

```
select title
       || ' was released in '
       || year_released movie_release
from movies
where country = 'us';
```

Although YEAR_RELEASED is actually a number, it's implicitly turned into a string by the DBMS.
- In that case it's not a big issue, but it would be better to use a function to convert explicitly.

```
select title
       || ' was released in '
       || cast(year_released as varchar) movie_release
from movies
where country = 'us';
```

Some Functions

- When to use functions
  - An example of showing a result that isn't stored as such is computing an age
    - You should never store an age; it changes all the time!
    - If you want to display the age of people who are alive, you must compute their age by subtracting the year when they were born from the current year.

Some Functions

- When to use functions
  - An example of showing a result that isn't stored as such is computing an age
    - You should never store an age; it changes all the time!
    - If you want to display the age of people who are alive, you must compute their age by subtracting the year when they were born from the current year.

  - In the table people:
    - Alive – died is `null`
    - Age: `<this year> - born`

```
select peopleid, surname,
       date_part('year', now()) - born as age
from people
where died is null;
```

7		7	Caroline	Aaron		1952	<null>	F
8		8	Quinton	Aaron		1984	<null>	M
9		9	Dodo	Abashidze		1924	1990	M

Some Functions

- Numerical functions

```
round(3.141592, 3)   -- 3.142
trunc(3.141592, 3)   -- 3.141
```

- More string functions

```
upper('Citizen Kane')
lower('Citizen Kane')
substr('Citizen Kane', 5, 3)  -- 'zen'
trim('  Oops  ')  -- 'Oops'
replace('Sheep', 'ee', 'i')  -- 'Ship'
```

Some Functions

- Type casting
  - cast(column as type)

```
select cast(born as char)||'abc' from people;
select cast(born as char(2)) ||'abc' from people;
select cast(born as char(10)) ||'abc' from people;
select cast(born as varchar) ||'abc' from people;
select cast(born as varchar(2)) ||'abc' from people;
```

Case

- A very useful construct is the CASE ... END construct that is similar to IF or SWITCH statements in a program
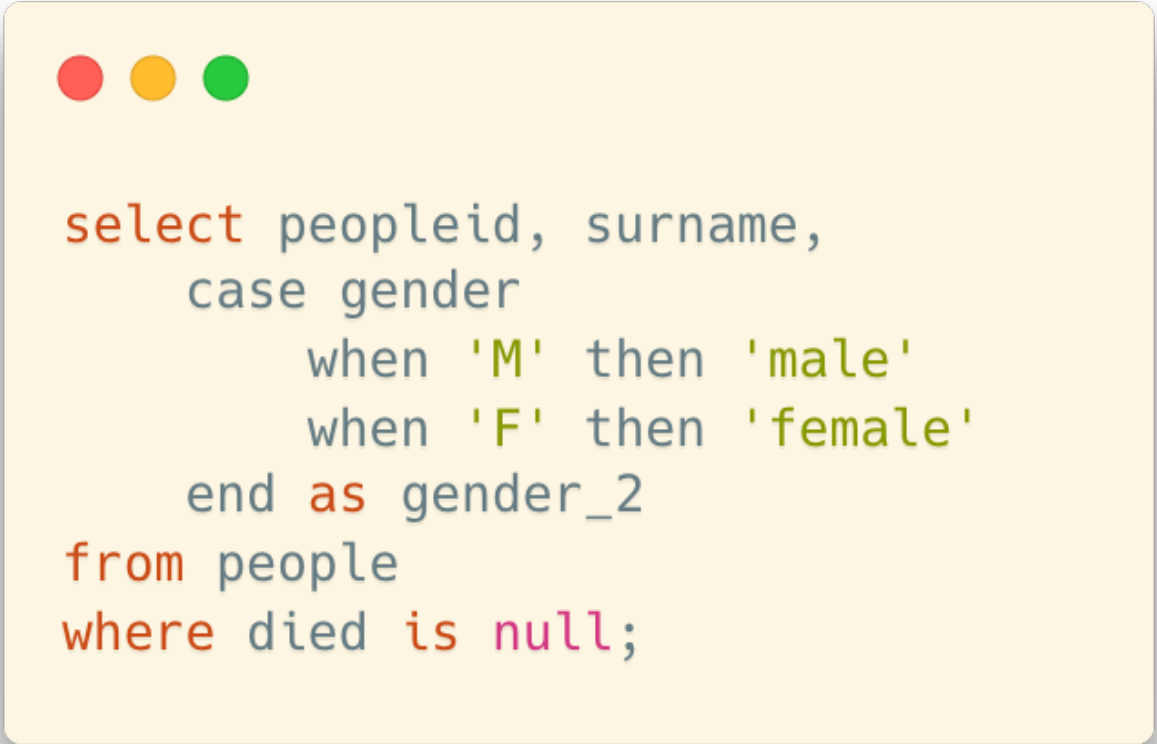
```
CASE input_expression
    WHEN when_expression THEN result_expression
    [ ...n ]
    [ELSE else_result_expression]
END
```

```
CASE
    WHEN Boolean_expression THEN result_expression
    [ ...n ]
    [ELSE else_result_expression]
END
```

Case

- Example 1: Show the corresponding words of the gender abbreviations

```sql
select peopleid, surname,
    case gender
        when 'M' then 'male'
        when 'F' then 'female'
    end as gender_2
from people
where died is null;
```

*Similar to the switch-case statement in Java and C

Case

- Example 2: Decide whether someone's age is older/younger than a pivot

Case

- Example 2: Decide whether someone's age is older/younger than a pivot

A horrible solution!

```
case age
    when 30 then 'younger than 44'
    when 31 then 'younger than 44'
    when 32 then 'younger than 44'
    when 33 then 'younger than 44'
    when 34 then 'younger than 44'
    when 35 then 'younger than 44'
    when 36 then 'younger than 44'

    ...

    when 43 then 'younger than 44'
    when 44 then '44 years old'
    when 45 then 'older than 44'

    ...
end as status
```

Case

- Example 2: Decide whether someone's age is older/younger than a pivot
  - CASE

```sql
select peopleid, surname,
    case (date_part('year', now()) - born > 44)
        when true then 'older than 44'
        when false then 'younger than 44'
        else '44 years old'
    end as status
from people
where died is null;
```

Case

- Example 2: Decide whether someone's age is older/younger than a pivot
  - CASE
  - CASE WHEN

```
select peopleid, surname,
    case
        when (date_part('year', now()) - born > 44) then 'older than 44'
        when (date_part('year', now()) - born < 44) then 'younger than 44'
        else '44 years old'
    end as status
from people
where died is null;
```

Case

- Example 2: Decide whether someone's age is older/younger than a pivot
  - CASE
  - CASE WHEN

```sql
select peopleid, surname,
    case
        when (date_part('year', now()) - born > 44) then 'older than 44'
        when (date_part('year', now()) - born < 44) then 'younger than 44'
        else '44 years old'
    end as status
from people
where died is null;
```

The ELSE branch
- Return a default value when all when criteria are not met
- If no else, NULL will be returned

Case

- About the NULL value
  - Use the "`is null`" criteria

```sql
select surname,
    case
        when died is null then 'alive and kicking'
        else 'passed away'
    end as status
from people
```