Using Advanced SQL

1. Revisiting the select command

The SELECT command format

[**WITH** [**RECURSIVE**] *with_query* [, ...]]
SELECT [**ALL** | **DISTINCT** [**ON** (*expression* [, ...])]] * | *expression* [[**AS**] *output_name*] [, ...]
[**FROM** *from_item* [, ...]]
[**WHERE** *condition*]
[**GROUP BY** *expression* [, ...]]
[**HAVING** *condition* [, ...]]
[**WINDOW** *window_name* AS (*window_definition*) [, ...]]
[{ **UNION | INTERSECT | EXCEPT** **}** **[** **ALL** | **DISTINCT**] *select*]
[**ORDER BY** *expression* [ASC | DESC | USING *operator*] [NULLS { FIRST | LAST }] [, ...]]
[**LIMIT** { *count* | ALL }]
[**OFFSET** *start* [ROW | ROWS]]
[**FETCH** { FIRST | NEXT } [*count*] { ROW | ROWS } ONLY]
[**FOR** { **UPDATE** | **SHARE** } [**OF** *table_name* [, ...]] [NOWAIT] [...]]

file:///C:/Program%20Files/PostgreSQL/9.4/doc/postgresql/html/sql-select.html

The DISTINCT Clause

- **[ALL | DISTINCT [ON (*expression* [,...])]]**
  - **ALL** = all records returned in the result set (default)
  - Default: **DISTINCT** only eliminates records that are complete duplicates
  - **SELECT DISTINCT ON** (*expression* [, ...]) keeps only the **first row** of each set of rows where the given expressions evaluate to equal.

  (The DISTINCT ON expressions are interpreted using the same rules as for ORDER BY; and *must match the leftmost* ORDER BY expression(s))

SELECT "City", "State" FROM store."Customer";

SELECT distinct on ("City") "City", "State" FROM store."Customer";

SELECT distinct on (“State") “State", “City" FROM store."Customer";

SELECT distinct on (“State") “State", “City" FROM store."Customer“ ORDER BY “State” DESC, “City” DESC;

The SELECT List

- *** | *expression* [AS *output_name*] [,...]**
  - AS option allows you to change the column heading label in the output to a value different from the column name
  - AS: optional, but it is recommended that you always either write AS or double-quote the output name (for protection against possible future keyword)
  - Output name: can be used to in ORDER BY and GROUP BY clauses, but not in the WHERE or HAVING clauses

SELECT *"CustomerID"* ***AS*** *"ID", "LastName" "Family","FirstName“ "Person"*

FROM *store."Customer";*

The FROM Clause

- **FROM *from_list* [,...]**
- The **most complex part** of the SELECT command
  - [ONLY] *table_name* [*] [[AS] *alias* [(*column_alias* [, ...])]]
  - (*select*) [AS] *alias* [(*column_alias* [, ...])]
  - *with_query_name* [[AS] *alias* [(*column_alias* [, ...])]]
  - *function_name* ([*argument* [, ...]]) [AS] *alias* [(*column_alias* [, ...] | *column_definition* [, ...])]
  - *function_name* ([*argument* [, ...]]) AS (*column_definition* [, ...])
  - *from_item* [NATURAL] *join_type from_item* [ON *join_condition* | USING (*join_column* [, ...])]

file:///C:/Program%20Files/PostgreSQL/9.4/doc/postgresql/html/sql-select.html#SQL-FROM

Standard Table Names

[ONLY] *table_name* [*] [[AS] *alias* [(*column_alias* [,...])]]

- ONLY option directs PostgreSQL to search only the table specified, and not any tables that inherit the specified table
- * directs PostgreSQL to search all child tables of the specified table

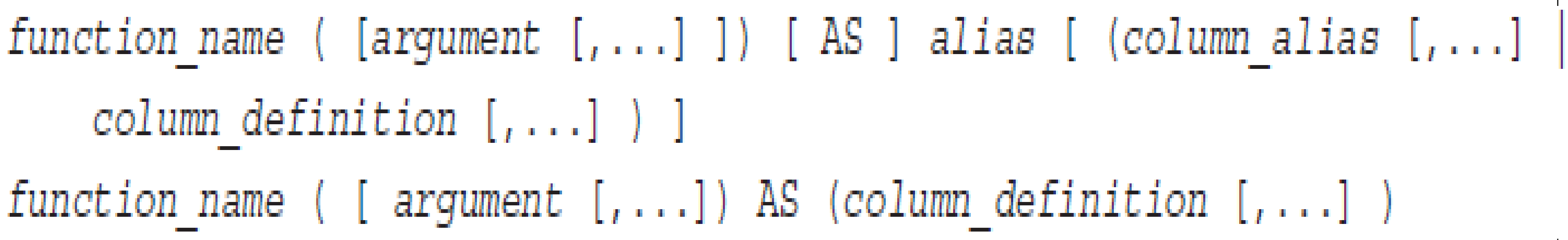
The Sub-select

- **(*select*) [AS] *alias* [(*column_alias* [,...])]**

```
SELECT *
FROM  (select "CustomerID", "FirstName"
             from store."Customer") as test ("ID",
"Name");
```

Functions

- The result set of the declared function is used as the input to the first SELECT command



```
function_name ( [argument [,...] ]) [ AS ] alias [ (column_alias [,...] |
    column_definition [,...] ) ]
function_name ( [ argument [,...]) AS (column_definition [,...] )
```

Joins

- ***from_item* [NATURAL] *join_type* *from_item* [ON *join_condition* | USING (*join_column* [,...])]**
  - NATURAL keyword is used to join tables on common column names
  - USING keyword to define specific matching column names in both tables
  - ON keyword to define a join condition
- ***join_type***: [INNER] JOIN, LEFT [OUTER] JOIN, RIGHT [OUTER] JOIN, FULL [OUTER] JOIN, CROSS JOIN

```
SELECT "Customer"."LastName", "Customer"."FirstName",
        "Product"."ProductName", "Order"."TotalCost"
FROM store."Order"
        NATURAL INNER JOIN store."Customer"
        NATURAL INNER JOIN store."Product";
```

Joins

```
SELECT "Customer"."LastName", "Customer"."FirstName",
        "Product"."ProductName", "Order"."TotalCost"
FROM  store."Order" INNER JOIN store."Customer" USING
   ("CustomerID")
        INNER JOIN store."Product" USING ("ProductID");


SELECT "Customer"."LastName", "Customer"."FirstName",
        "Product"."ProductName", "Order"."TotalCost"
FROM store."Order" INNER JOIN store."Customer" ON
   (store."Order"."CustomerID" = store."Customer"."CustomerID")
        INNER JOIN store."Product" USING ("ProductID");
```

The WHERE Clause

- **WHERE *condition* [,...]**

The GROUP BY Clause

- We use a GROUP BY clause to group tuples, following WHERE clause
- Syntax:

  GROUP BY grouping attributes or *expression* [,...]
- GROUP BY clause be always used with a PostgreSQL function that aggregates values from similar records

```
SELECT  "Product"."ProductID", "ProductName ", sum("Quantity"),
FROM store."Order"  NATURAL INNER JOIN store."Product"
GROUP BY "Product"."ProductID";
```

- Be careful with the GROUP BY and ORDER BY clauses
  - GROUP BY clause groups similar records BEFORE the rest of the SELECT command is evaluated
  - ORDER BY clause orders records AFTER the SELECT commands are processed

The HAVING Clause

- **HAVING *condition* [,...]**
- The HAVING clause is similar to the WHERE clause, in that it is used to define a filter condition to limit records used in the GROUP BY clause

- Records that do not satisfy the WHERE conditions are not processed by the GROUP BY clause

- If there is no GROUP BY clause, the presence of HAVING turns all the selected rows as a single group.

file:///C:/Program%20Files/PostgreSQL/9.4/doc/postgresql/html/sql-select.html#SQL-HAVING

The Set Operation Clauses

- ***select1* [(UNION | INTERSECT | EXCEPT]) [ALL] *select2***
- The Set Operation clause types are
  - UNION Display all result set records in both *select1* and *select2*
  - INTERSECT Display only result set records that are in both *select1* and *select2*
  - EXCEPT Display only result set records that are in *select1* but not in *select2*
- By default, **duplicate record**s in the output set **are not displayed** → ALL

The ORDER BY Clause

- [ORDER BY *expression* [ASC | DESC | USING *operator*] [NULLS { FIRST | LAST }] [,...]]
- By default, the ORDER BY clause orders records in ascending order
- The USING parameter declares an alternative operator to use for ordering
  - (<) is equivalent to the ASC keyword
  - (>) is equivalent to the DESC keyword
- NULLS LAST: null values sort after all non-null values
- NULLS FIRST: null values sort before all non-null values

The LIMIT Clause

- **[LIMIT (*count* | ALL)] [OFFSET *start*]**
- The LIMIT clause specifies a maximum number of records to return in the result set
- The default behavior is LIMIT ALL, which returns all records in the result set
- The OFFSET parameter allows you to specify the number of result set records to skip before displaying records in the output
  - first record in the result set is at *start* value 0
  - *Start* value 1 is the second record

```
SELECT * FROM store."Customer"
ORDER BY "CustomerID" LIMIT 3 OFFSET 1;
```

The FOR Clause

- **[FOR (UPDATE | SHARE) [OF *table_name* [,...] [NOWAIT]]**

- FOR UPDATE locks the records (viewing, deleting, or modifying)
- NOWAIT parameter, the SELECT command does not wait, but instead exits with an error stating that the records are locked
- FOR SHARE clause allows other users to view the records
- If you do not want to lock all of the records returned in the result set, combine the FOR clause with the LIMIT clause

- More details:

  file:///C:/Program%20Files/PostgreSQL/9.4/doc/postgresql/html/sql-select.html#SQL-FOR-UPDATE-SHARE

The WITH Clause

- **[WITH** [**RECURSIVE**] *with_query* [, ...]**]**

- WITH provides a way to write auxiliary statements for use in a larger query
- Each auxiliary statement in a WITH clause can be a SELECT, INSERT, UPDATE, or DELETE;
- All queries in the WITH list are computed ➔ temporary tables that can be referenced in the FROM list. A WITH query that is **referenced more than once** in FROM is **computed only once**.

file:///C:/Program%20Files/PostgreSQL/9.4/doc/postgresql/html/sql-select.html#SQL-WITH

The WITH Clause

```
WITH customer_totalorder AS(
   SELECT "CustomerID", sum("Quantity") as total_quantity
   FROM store."Order"
   GROUP BY "CustomerID")
SELECT * FROM store."Customer"
WHERE "CustomerID" IN
          (SELECT "CustomerID"
          FROM customer_totalorder
          WHERE total_quantity = (SELECT
   MAX(total_quantity)
                                             FROM customer_totalorder) );
```

The WITH RECURSIVE

- **[WITH** [**RECURSIVE**] *with_query* [, ...]**]**
  - If RECURSIVE is specified, it allows a SELECT subquery to reference itself by name. Such a subquery must have the form:

    non_recursive_term

    UNION [ALL | DISTINCT] *recursive_term*

The WITH RECURSIVE

```
WITH RECURSIVE tmp_table(n) AS (
        values (1)
  UNION ALL
        SELECT n+1 FROM tmp_table WHERE n
  <10)
SELECT * FROM tmp_table;
```

WITH RECURSIVE: Recursive queries are typically used to deal with hierarchical or tree-structured data

The WITH RECURSIVE

```
\c test
CREATE TABLE sujects (
            sid char(5) primary key,
            sname varchar(20),
            scredits int,
            sid_required char(5) );
INSERT INTO sujects VALUES
        ('IT010', 'Trí tuệ nhân tạo', 3, 'IT005'),
        ('IT005', 'Cau truc DL va GT', 2, 'IT001'),
        ('IT001', 'Tin hoc dai cuong', 2, NULL),
        ('IT006', 'CSDL', 3, 'IT001');
```

The WITH RECURSIVE

```
WITH RECURSIVE sujects_required(sid, sid_required)
   AS (
         SELECT sid, sid_required
         FROM sujects
         WHERE sid = 'IT010'
   UNION ALL
         SELECT s1.sid, s1.sid_required
         FROM sujects s1, sujects_required s2
         WHERE s1.sid = s2.sid_required)
SELECT * FROM sujects_required;
```