DATABASES

Lecture 4. DML. Inserting, updating, deleting data

Lexical Structure

- SQL input consists of a sequence of commands.

SELECT * FROM employees; INSERT INTO employees (name, salary) VALUES ('John', 50000);

- A command is composed of a sequence of tokens, terminated by a semicolon (“;”)

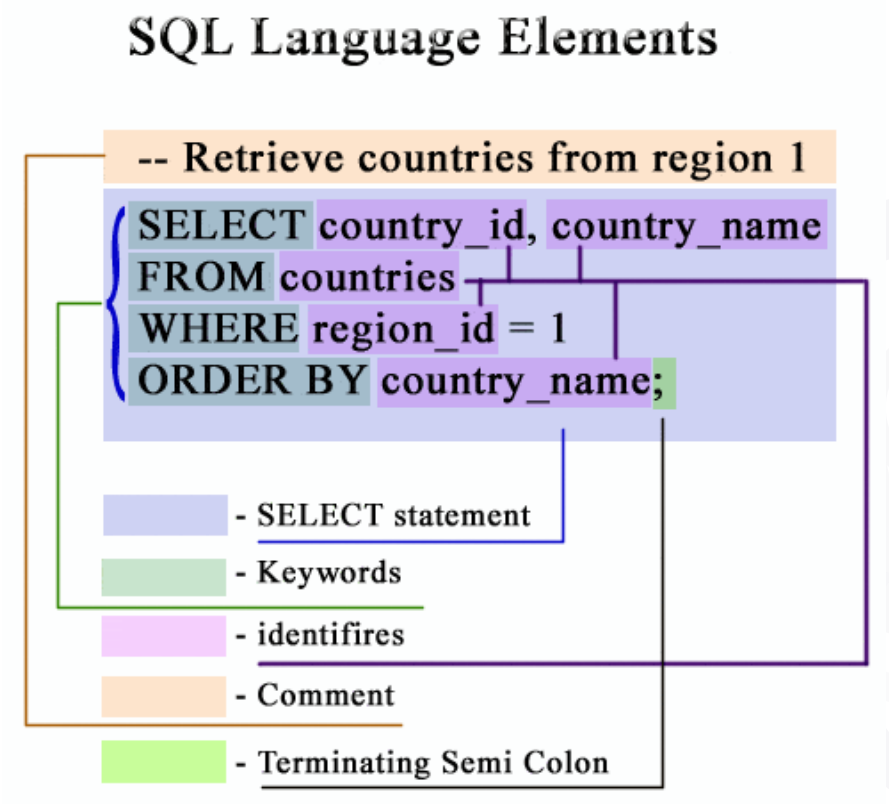
SELECT name, age FROM students WHERE age > 18;

- The end of the input stream also terminates a command

SELECT * FROM employees

Lexical Structure

- A token can be a keyword (SELECT, FROM, INSERT, WHERE, AND, OR, JOIN), an identifier, a quoted identifier, a literal (or constant), or a special character symbol
- Which tokens are valid depends on the syntax of the particular command
- More than one command can be on a line.



SELECT * FROM employees; INSERT INTO employees (name, age) VALUES ('Alice', 30); UPDATE employees SET age = 31 WHERE name = 'Alice';

Lexical Structure

- Commands can usefully be split across lines

```
SELECT name, age
FROM employees
WHERE age > 30
ORDER BY name;
```

- Comments can occur in SQL input. They are not tokens, they are effectively equivalent to whitespace

Identifiers and Keywords naming

- SQL identifiers must begin with a letter or an underscore.

```
my_table                    1column -- Starts with a digit
column1                     #table -- Starts with a special character
_employee_name
```

- SQL standard will not define a keyword that contains digits or starts or ends with an underscore.

```
SELECT                      SEL1CT
INSERT                      _SELECT_
DELETE                      INSERT2
```

- Maximum identifier length is 63 bytes.
- Key words and unquoted identifiers are case insensitive, but it is better to write key words in upper and identifiers in lower case. But "Employee Name" is different from "employee name“.

```
SELECT name FROM employees WHERE age > 30;  -- Best practice
```

Operator Precedence (highest to lowest)

Operator/Element	Associativity	Description
.	left	table/column name separator
::	left	PostgreSQL-style typecast
[ ]	left	array element selection
+ -	right	unary plus, unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
(any other operator)	left	all other native and user-defined operators
BETWEEN IN LIKE ILIKE SIMILAR		range containment, set membership, string matching
< > = <= >= <>		comparison operators
IS ISNULL NOTNULL		IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM, etc
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

Operator Precedence (examples)


```
SELECT *
FROM students
WHERE age > 18 OR age < 10 AND active = true;
```

```
SELECT *
FROM students
WHERE (age > 18 OR age < 10) AND active = true;
```

Operator Precedence (examples)

```
SELECT * FROM employees
WHERE department = 'HR' OR department = 'IT'
AND salary > 50000;
```

```
SELECT * FROM employees
WHERE (department = 'HR' OR department = 'IT')
AND salary > 50000;
```

Operator Precedence (examples)


```
SELECT * FROM students
WHERE NOT active OR age < 20 AND city =
'London';
```

```
SELECT * FROM students
WHERE NOT (active OR age < 20) AND city =
'London';
```

DML - Data Manipulation Language

DML – Data Manipulation Language (SELECT, INSERT, UPDATE, DELETE). DML statements are SQL statements that manipulate data.

- **SELECT** is used to query data from one or more tables
- **INSERT INTO** is used to add new rows to a table.
- **UPDATE** is used to modify existing rows in a table
- **DELETE** removes rows from a table based on a condition.

Inserting Data

The INSERT INTO statement is used to add new records into a database table

- When a table is created, it contains no data
- The first thing to do before a database can be of much use is to insert data
- Even if you know only some column values, a complete row must be created

```
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
```

Inserting Data. Parameters


```
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
```

- **table_name** - The name (optionally schema-qualified) of an existing table.
- **alias** - A substitute name for table_name. When an alias is provided, it completely hides the actual name of the table.
- **column_name** - The name of a column in the table named by table_name.
- **DEFAULT VALUES** - All columns will be filled with their default values.
- **expression** - An expression or value to assign to the corresponding column.
- **DEFAULT** - The corresponding column will be filled with its default value.
- **query** - A query (SELECT statement) that supplies the rows to be inserted.

Inserting Data

- The target column names can be listed in any order.

```
INSERT INTO employees (salary, name, department)
    VALUES (55000, 'Alice', 'HR');
```

- If no list of column names is given at all, the default is all the columns of the table in their declared order.

```
INSERT INTO employees
    VALUES ('Alice', 'HR', 55000);
```

- Or the first N column names, if there are only N columns supplied by the VALUES clause or query.

```
INSERT INTO employees VALUES ('Alice', 'HR');
```

Inserting Data

- If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

```
INSERT INTO employees (name, department, salary)
        VALUES ('Carol', 'Finance', '55000');
```

- You must have INSERT privilege on a table to insert into it.

```
GRANT INSERT ON employees TO user1;
```

Inserting Data

```
INSERT INTO films VALUES
            ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

```
INSERT INTO films DEFAULT VALUES;
```

Inserting Data

Example

```
INSERT INTO films
    SELECT * FROM tmp_films
        WHERE date_prod < '2004-05-07';
```

Updating Data

UPDATE — update rows of a table

- You can update individual rows, all the rows in a table, or a subset of all rows

```
UPDATE employees
SET salary = 60000
WHERE department = 'HR';
```

```
UPDATE employees
SET salary = 60000;
```

- Each column can be updated separately. The other columns are not affected.
- UPDATE changes the values of the specified columns in all rows that satisfy the condition
- Only the columns to be modified need be mentioned in the SET clause
- Columns not explicitly modified retain their previous values.

Updating Data. Parameters

```
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] ) |
          ( column_name [, ...] ) = ( sub-SELECT )
        } [, ...]
    [ FROM from_list ]
    [ WHERE condition ]
```

- **table_name** - The name (optionally schema-qualified) of the table to update.
- If **ONLY** is specified before the table name, matching rows are updated in the named table only. If ONLY is not specified, matching rows are also updated in any tables inheriting from the named table.
- Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

Updating Data

- **alias** - A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table.
- **column_name** - The name of a column in the table named by table_name.
- **expression** - An expression to assign to the column. The expression can use the old values of this and other columns in the table.
- **DEFAULT** - Set the column to its default value (which will be NULL if no specific default expression has been assigned to it).
- **sub-SELECT** - A SELECT sub-query that produces as many output columns as are listed in the parenthesized column list preceding it. The sub-query must yield no more than one row when executed.
- **from_list** - A list of table expressions, allowing columns from other tables to appear in the WHERE condition and the update expressions.
- **condition** - An expression that returns a value of type boolean. Only rows for which this expression returns true will be updated.

Updating Data

```
UPDATE films SET kind = 'Dramatic';
```

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';

UPDATE weather SET temp_lo = temp_lo+1, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

```
UPDATE weather SET (temp_lo, prcp) = (temp_lo+1, DEFAULT)
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Deleting Data

DELETE — delete rows of a table

- You can only remove entire rows from a table
- Removing rows can only be done by specifying conditions that the rows to be removed have to match
- You can remove all rows in the table at once
- DELETE deletes rows that satisfy the WHERE clause from the specified table.
- If the WHERE clause is absent, the effect is to delete all rows in the table.

Single row

```
DELETE FROM student WHERE ID = 41;
```

Multiple rows

```
DELETE FROM student WHERE gpa < 3.6;
```

All rows

```
DELETE FROM student;
```

Deleting Data

```
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

- **using_list** - A list of table expressions, allowing columns from other tables to appear in the WHERE condition.

```
/*
This deletes all rows in the employees table where the employee's
department matches the 'Marketing' department in the
departments table.  */
DELETE FROM employees e
USING departments d
WHERE e.department_id = d.id
AND d.department_name = 'Marketing';
```

- **condition** - An expression that returns a value of type boolean. Only rows for which this expression returns true will be deleted.

Foreign Key Constraints: When deleting rows from a table that has foreign key constraints, PostgreSQL enforces referential integrity. If the row being deleted has dependent rows in other tables, it might be necessary to define an ON DELETE rule (like CASCADE, RESTRICT, etc.).

Deleting Data

```sql
DELETE FROM films;
```

```sql
DELETE FROM films WHERE kind <> 'Musical';
```

```sql
DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name = 'foo';
```

Returning Data From Modified Rows

- Sometimes it is useful to obtain data from modified rows while they are being manipulated.
- The INSERT, UPDATE, and DELETE commands all have an optional RETURNING clause that supports this.

```
INSERT INTO table_name (...) VALUES (...) RETURNING expression;
UPDATE table_name SET ... WHERE condition RETURNING expression;
DELETE FROM table_name WHERE condition RETURNING expression;
```

- Use of RETURNING avoids performing an extra database query to collect the data
- The allowed contents of a RETURNING clause are the same as a SELECT command's output list
- A common shorthand is RETURNING *, which selects all columns of the target table in order.

Returning Data From Modified Rows


```
INSERT INTO users (firstname, lastname)
    VALUES ('Joe', 'Cool')
    RETURNING id;
```

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

Returning Data From Modified Rows


```
DELETE FROM products
  WHERE obsoletion_date = 'today'
  RETURNING *;
```