4. Managing Tables and Views

Introduction to PostgreSQL

AGENDA

- Basic table creation
- Table constraints
- Table relationships
- Referential integrity
- Designing tables well

TABLE MANAGEMENT

- Tables and related objects are created in PostgreSQL using standard SQL commands
- Tables are created in a schema in a database
  - If no schema is specified, then the table is created in the default public schema
- Tables are owned by the user that created them
  - Access privileges are given by the owner to other users with GRANT

TABLE CREATION

- Created using CREATE TABLE
  - List tables with \dt
  - List table details with \d <tablename>
- Delete tables with DROP TABLE
  - All data in the table is lost
  - Tables can only be dropped by the owner or user with permission to drop the table

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);

CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

```
lab4=# DROP TABLE cities, weather;
lab4=# \dt
      List of relations
 Schema | Name | Type | Owner
--------+------+------+-------
(0 rows)
```

CREATE TABLE AS

- Create new table by copying the data and structure of the existing table.
  - WITH NO DATA only copies the structure
  - Otherwise the data is copied
- SELECT statements can be used to populate the new table with a subset of the original data

```
lab4=# SELECT * FROM employees;
 id |  name   | department | salary
----+---------+------------+--------
  1 | Alice   | Sales      |  50000
  2 | Bob     | Marketing  |  55000
  3 | Charlie | Sales      |  60000
  4 | David   | IT         |  65000
  5 | Eve     | HR         |  48000
(5 rows)


lab4=# CREATE TABLE ecopy AS TABLE employees;
lab4=# SELECT * FROM ecopy;
 id |  name   | department | salary
----+---------+------------+--------
  1 | Alice   | Sales      |  50000
  2 | Bob     | Marketing  |  55000
  3 | Charlie | Sales      |  60000
  4 | David   | IT         |  65000
  5 | Eve     | HR         |  48000
(5 rows)
```

TABLE INHERITANCE

- In the OO view, tables represent types of object
  - We can create sub-types that inherit from base types
- The sub-type
  - Has the same columns as its parent
  - Generally has more attributes or columns
- This is used for table design rather than storing normalized data.
  - Working with data across multiple child tables can be complex
  - Often partitioning is a better solution

```sql
CREATE TABLE cities (
    name        text,
    population  float8,
    elevation   int     -- (in ft)
);

CREATE TABLE capitals (
    state       char(2)
) INHERITS (cities);

-- Now, let's populate the tables.
INSERT INTO cities VALUES ('San Francisco', 7.24E+5, 63);
INSERT INTO cities VALUES ('Las Vegas', 2.583E+5, 2174);
INSERT INTO cities VALUES ('Mariposa', 1200, 1953);

INSERT INTO capitals VALUES ('Sacramento', 3.694E+5, 30, 'CA');
INSERT INTO capitals VALUES ('Madison', 1.913E+5, 845, 'WI');

SELECT * FROM cities;
SELECT * FROM capitals;
```

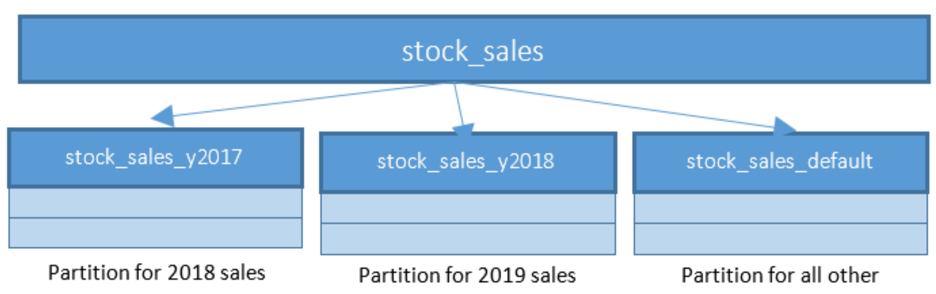
TABLE PARTITIONING

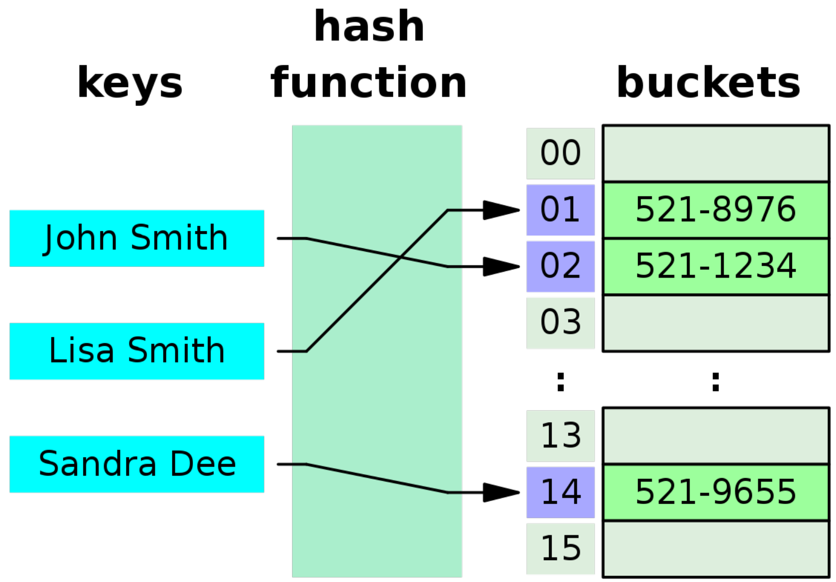
- Partitioning divides large tables into smaller, more manageable pieces
  - Different methods for partitioning are:
  - Range – data is grouped by ranges of values for a specific attribute, like transaction data
  - List – similar to range but on a categorical attribute, like county name for example
  - Hash – uses a hash c part partitioning methods. Here's an example of how to create a partitioned table using range partitioning:

```
CREATE TABLE cities (
    name         text,
    population   float8,
    elevation    int      -- (in ft)
);

CREATE TABLE capitals (
    state        char(2)
) INHERITS (cities);

-- Now, let's populate the tables.
INSERT INTO cities VALUES ('San Francisco', 7.24E+5, 63);
INSERT INTO cities VALUES ('Las Vegas', 2.583E+5, 2174);
INSERT INTO cities VALUES ('Mariposa', 1200, 1953);

INSERT INTO capitals VALUES ('Sacramento', 3.694E+5, 30, 'CA');
INSERT INTO capitals VALUES ('Madison', 1.913E+5, 845, 'WI');

SELECT * FROM cities;
SELECT * FROM capitals;
```

TABLE PARTITIONING

- Partitioning divides large tables into smaller, more manageable pieces
  - Range partitioning – data is grouped by ranges of values for a specific attribute, like transaction data
  - List – similar to range but on a categorical attribute, like county name for example
  - Hash – uses hashing into partitions based on hashing a specific attribute
    - *Provides a more uniform distribution of data if the data is skewed along a partition attribute*
    - *We define the number of partitions*
    - *Useful with large data sets because it improves the performance of queries*





TABLESPACES

- Tablespaces define locations on the file system where the database objects are stored.
  - By default all data is stored in the default tablespace located in the data directory.
  - Tablespaces define other locations for storing data by mapping it to specific physical locations.
  - Created by defining a directory PostgreSQL has permission to read and write data files.
- Benefits
  - By distributing data across different physical disks, I/O performance can be improved.
  - New tablespaces on different disks can expand physical storage without moving the entire database.
  - Allows DBAs to specify where specific tables, indexes, or entire schemas are stored, making it easier to manage large and complex databases.
  - Provides a way to separate data physically.
  - Streamlines backup and recovery

TABLESPACES

- Possible issues
  - Can add complexity administration because of track of where different object are located.
  - For physical backups tablespaces must be backed up and restored to their respective locations or errors could result.
  - Logical backups don’t capture tablespace information by default
- pg_default Tablespace
  - Where user defined database are stored by default unless specified otherwise.
- pg_global Tablespace
  - Used for storing shared system catalogs and objects that are shared across the entire cluster.
    - *Objects like pg_database, which keeps track of all databases in the cluster, and pg_authid, which stores role and authentication information. T*
  - Not accessible for user defined objects

VIEWS

- Virtual tables that represent the result of a query.
  - The query is stored in raw SQL format so it can be reused
  - Used to simplify complex queries, encapsulate business logic, and present data without duplicating the actual data in the underlying tables
- Limitations of Views:
  - Views execute their underlying query each time they are accessed
  - Complex views can have performance issues, especially if there are multiple joins or large datasets.
- Not all views are updateable
  - Updateable means that changes made to the view are propagated into the tables
  - Updateable must meet specific criteria, such as being based on a single table without groupings or aggregates.

VIEWS

- Updateable view requirements
  - The view must be defined by a simple SELECT statement that meets the following criteria:
    - *It selects from a single table.*
    - *It does not include any aggregation (GROUP BY, HAVING).*
    - *It does not include DISTINCT, set operations (UNION, INTERSECT, EXCEPT), or subqueries.*
    - *It does not include window functions.*
    - *It does not use LIMIT or OFFSET.*
    - *It does not contain JOIN clauses.*
  - Each column in the view must directly map to a column in the underlying base table without any expressions, computations, or column aliases that modify data types or values.
  - The underlying table should have a primary key or a unique constraint to uniquely identify rows.

MATERIALIZED VIEWS

- Similar to regular views
  - Unlike regular views, materialized views store the result set of the query on disk, allowing for faster access to the data.
  - Since the data is stored, materialized views need to be refreshed to reflect changes in the underlying tables.
  - Refreshing can be done manually or automatically.
  - Ideal for queries that are expensive to run and do not need real-time data accuracy, such as reporting or analytical queries.

DATA TYPES

- PostgreSQL supports the standard SQL data types
  - Also supports a number of complex data types
  - This will be reviewed in the documentation
- Default values can be specified
  - Either values of some computation
  - Like the ‘now()’ function

CREATING DATA TYPES

- PostgreSQL also allows user to create custom data types
- Common custom types
  - Composite Types: allow grouping multiple fields of different data types into a single logical unit.
  - Enum Types: static, predefined values for attributes with a limited set of possible values
    - *Excellent for enforcing referential integrity*
  - Domain Types: on existing data types but include constraints
    - *CREATE DOMAIN positive_int AS INT CHECK (VALUE > 0);*

```
CREATE TYPE full_name AS (
    first_name TEXT,
    last_name TEXT
);

CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    name full_name,
    position TEXT
);

INSERT INTO employees (name, position) VALUES
(('John', 'Doe'), 'Software Developer'),
(('Jane', 'Smith'), 'Project Manager');



SELECT name.first_name FROM employees;
```

CUSTOM DATA TYPES

- Benefits
  - Maintain data integrity by enforcing consistent data structures and validation rules.
  - Encapsulating related fields into a single type allows for simpler queries
  - Promotes re-usability across multiple tables and applications, reducing redundancy.
  - User-defined types makes PostgreSQL highly extensible, for specific application needs.
- Considerations
  - Custom types can increase schema complexity
  - Some custom types, especially those requiring complex operations, might impact performance.
  - Might introduce compatibility challenges when interfacing with other systems or migrating data.

LAB 4-1

- The lab description and documentation is in the Lab directory in the class repository

TABLE CONSTRAINTS

- Table constraints are rules applied to columns and tables
  - Enforce data integrity, ensure consistency, and define the relationships between data.
  - Help prevent invalid data from being entered into the database
    - *Eg. Two customers with the same customer number*
  - Provide safeguards that maintain the logical correctness of the data.
    - *Eg. Invoice for a non-existent customer*
- Types of constraints
  - UNIQUE: ensures all values in a column or group of columns are unique across the table.
  - NOT NULL: Ensures that a column cannot have NULL values.
  - PRIMARY KEY: Combination of UNIQUE and NOT NULL
    - *Each table can have at most one primary key*
    - *Often a single column but can be combination of columns – called a composite key*

TABLE CONSTRAINTS

- FOREIGN KEY: Enforces a link between the data in two tables for referential integrity
  - *References the primary key or unique constraint of another table.*
  - *Prevents actions that would Invalidate the link like deleting the foreign table row if there are dependent rows in the table where the foreign key constraint is defined*
  - *Can specify actions on update or delete (e.g., CASCADE, SET NULL, RESTRICT)*
- CHECK: Enforces custom conditions that must be true for each row in the table.
  - *Similar to constraints when defining a domain.*
- EXCLUSION: Ensures that, for any two rows, the specified columns do not have overlapping values
  - *Commonly used with range and geometric data types.*
  - *Useful for scenarios like scheduling where no two time periods should overlap.*

TABLE CONSTRAINTS

- Constraints ensure that the data entered into a table meets specified criteria, maintaining data integrity and reliability.
- Constraints can also impact performance, especially on large tables with frequent updates.
  - Proper indexing and query optimization are necessary to mitigate performance degradation
- Constraints can be combined to enforce complex rules.
  - For instance, a column can be both NOT NULL and UNIQUE
- Some constraints, such as foreign keys, declared DEFERRED
  - This means they they are checked at the end of a transaction rather than immediately

RELATIONAL CONSTRAINTS

- The primary use of a relational constraint
  - Defines what should happen to a row in a child table that has a foreign key (parent table)
  - Changes in the parent table can invalidate the relationship, not the data
    - *For example, the customer table has a reference to the sales person table. What happens when a sales person if fired?*
- The changes that can affect referential integrity are DELETE and UPDATE for the parent table
- CASCADE
  - DELETE CASCADE: Deleting a row in the parent table deletes all matching rows in the child table
  - UPDATE CASCADE: Updating a primary key value in the parent table updates the corresponding foreign key values in the child table
  - Useful when the child records are entirely dependent on the parent record

RELATIONAL CONSTRAINTS

- RESTRICT
  - DELETE RESTRICT: Blocks deletion of a row in the parent table if a child table has any matching foreign key references
  - UPDATE RESTRICT: Blocks updating the primary key in the parent table if a child table has any matching foreign key references
  - Enforces the integrity of the child data by ensuring it cannot be removed or updated indirectly.
- SET NULL
  - DELETE SET NULL: Deleting a row in the parent table causes the corresponding foreign key values in the child table to be set to NULL.
  - UPDATE SET NULL: Updating a primary key in the parent table causes the corresponding foreign key values in the child table to be set to NULL.
  - Used when the child record should remain but the relationship to the parent should be removed

RELATIONAL CONSTRAINTS

- SET DEFAULT
  - DELETE SET DEFAULT: When a row in the parent table is deleted, the corresponding foreign key values in the child table are set to a predefined default.
  - UPDATE SET DEFAULT: When a primary key in the parent table is updated, the corresponding foreign key values in the child table are set to the default value.
  - Useful when a default relationship should be established if the original relationship is removed.
    - *Eg. The default sales person is the sales manager, when deleting a sales persons, all their accounts are transferred to the sales manager*
- NO ACTION
  - Any attempt to delete or update a row in the parent table that is referenced by a child row will result in an error.
  - Similar to RESTRICT, but the check is deferred until the end of the transaction, allowing for temporary inconsistencies within a transaction that can be fixed up during the transaction

RELATIONAL CONSTRAINTS

- RESTRICT
  - DELETE RESTRICT: Blocks deletion of a row in the parent table if a child table has any matching foreign key references
  - UPDATE RESTRICT: Blocks updating the primary key in the parent table if a child table has any matching foreign key references
  - Enforces the integrity of the child data by ensuring it cannot be removed or updated indirectly.
- CASCADE
  - DELETE CASCADE: Deleting a row in the parent table deletes all matching rows in the child table
  - UPDATE CASCADE: Updating a primary key value in the parent table updates the corresponding foreign key values in the child table
  - Useful when the child records are entirely dependent on the parent record

TRIGGERS

- Database objects that automatically execute in response to certain events
  - Help automate complex database logic, enforce constraints, maintain audit trails, and keep data consistent.
  - Fire when specific events (like INSERT, UPDATE, or DELETE) occur on a table or view.
  - Can be set to fire BEFORE or AFTER the triggering event, or even INSTEAD OF in the case of views.
  - Row-level triggers execute once for each row affected by the triggering event.
  - Statement-level triggers execute once per SQL statement, regardless of the number of rows affected.
- Triggers require a trigger function, which contains the actions or logic to be executed.
  - Written in PL/pgSQL or other supported languages.

LAB 4-2

- The lab description and documentation is in the Lab directory in the class repository

DESIGNING TABLES WELL

- Understand the Data and Its Usage Patterns
  - Use a data model to understand what the data represents and what constitutes valid data
  - Understand what will data accessed and used.
  - Consider the types of queries that will be run, how often the data will be updated, and the relationships between different entities.
- Choose Appropriate Data Types
  - Use the most suitable data types for each column to optimize storage and performance.
    - *For example, use INTEGER instead of BIGINT if the range of values allows it, or VARCHAR(n) with a defined length instead of TEXT when the length is predictable.*
  - Leverage PostgreSQL specific data types like JSONB for JSON data, ARRAY for lists, and UUID for unique identifiers.

DESIGNING TABLES WELL

- Normalize The Data
  - Apply normalization up to the third normal for to reduce redundancy
  - Avoid over-normalization, which can lead to excessive joins and performance issues.
  - Strike a balance between normalization and practical performance needs based on profiling performance
- Define Primary Keys
  - Ensure every table has a primary key that uniquely identifies each row.
  - Use SERIAL or BIGSERIAL types for auto-incrementing primary keys, or use UUID for distributed systems where uniqueness across databases is needed.
- Use Constraints to Enforce Data Integrity
  - Utilize constraints such as NOT NULL, UNIQUE, PRIMARY KEY, CHECK, and FOREIGN KEY to enforce rules and maintain data integrity.

DESIGNING TABLES WELL

- Use Indexes
  - Create indexes on columns that are frequently used in WHERE clauses, joins, and sorting operations to improve query performance.
  - Use appropriate index types: B-tree indexes for general-purpose indexing, GIN indexes for full-text search or JSONB data, and GiST indexes for geometric data.
  - Avoid over-indexing because indexes consume storage and can slow down write operations like INSERT, UPDATE, and DELETE.
- Partition Large Tables
  - Consider partitioning large tables to improve performance and manageability based on a partition key that is consistent with domain logic
  - Partitioning can reduce query times by scanning only relevant partitions and can improve maintenance tasks like vacuuming and backups.

DESIGNING TABLES WELL

- Optimize for Read and Write Patterns
  - Analyze how data will be accessed and optimized for the most frequent read and write operations.
  - For read-heavy workloads, consider denormalization or using materialized views to simplify and speed up complex queries.
  - For write-heavy workloads, minimize the number of indexes and avoid complex constraints that can slow down data modifications.
- Avoid NULLs Where Possible
  - Avoid excessive use of NULL values, which can complicate queries and affect indexing.
  - Use default values or redesign schemas to minimize the presence of NULLs, especially in indexed columns.

DESIGNING TABLES WELL

- Use Foreign Keys and Define Relationships Clearly
  - Use foreign keys to enforce referential integrity between tables, ensuring that relationships between data remain valid.
  - Use ON DELETE and ON UPDATE actions (CASCADE, SET NULL, RESTRICT) to specify what should happen when related data is modified.
- Data Security and Access Control
  - Plan for security by defining roles and permissions that restrict access to sensitive data.
  - Use views to provide controlled access to data and prevent direct access to base tables when necessary.
- Document the Schema
  - Maintain clear documentation for your schema, including descriptions of tables, columns, constraints, and the purpose of each.
  - Use descriptive names for tables, columns, and constraints to make the schema self-explanatory.

DESIGNING TABLES WELL

- Plan for Scalability and Growth
  - Design tables with future growth in mind, considering how the database might need to scale in terms of data volume and concurrent access.
  - Use sequences or UUIDs for keys that need to scale across distributed systems.
- Regularly Analyze and Vacuum Tables
  - Use the ANALYZE command to update statistics for the query planner, which helps optimize query execution.
  - Regularly run VACUUM to reclaim storage from deleted or updated rows and to prevent table bloat.

DESIGNING TABLES WELL

- Monitor Performance and Adjust
  - Continuously monitor database performance using PostgreSQL tools like pg_stat_activity, pg_stat_user_tables, and the EXPLAIN command to analyze query plans.
  - Make adjustments based on performance metrics, such as adding or removing indexes, adjusting table design, or modifying queries.

“Develop a passion for learning.”

End Module