

ACCA7002 Database Systems

Tutorial 11

Working with different types of indexes in PostgreSQL

Exercise 11.1: Primary Index

Objective: Create a table with a primary index, insert data, and observe the impact on performance.

```
-- Create a table with a primary index
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    salary INTEGER
);

-- Insert sample data
INSERT INTO employees (first_name, last_name, salary) VALUES
    ('John', 'Doe', 50000),
    ('Jane', 'Smith', 60000),
    ('Bob', 'Johnson', 75000);

-- Query data using the primary key
SELECT * FROM employees WHERE employee_id = 2;
```

Exercise 11.2: Clustered Index

Objective: Create a table, cluster it based on an index, and observe the impact on query performance.

```
-- Create a table
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INTEGER,
    order_date DATE,
    total_amount DECIMAL(10, 2)
);

-- Insert sample data
INSERT INTO orders (customer_id, order_date, total_amount)
VALUES
    (1, '2023-01-15', 120.50),
    (2, '2023-02-10', 200.00),
    (1, '2023-03-05', 75.25);
```

```
-- Create an index on customer_id
CREATE INDEX idx_customer ON orders(customer_id);

-- Cluster the table based on the created index
CLUSTER orders USING idx_customer;

-- Query data based on the clustered index
SELECT * FROM orders WHERE customer_id = 1;
```

Exercise 11.3: Secondary Index

Objective: Create a table with a secondary index, insert data, and observe the impact on query performance. Modify the queries, and observe the performance differences.

```
-- Create a table with a secondary index
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10, 2)
);

-- Create a secondary index on the category column
CREATE INDEX idx_category ON products(category);

-- Insert sample data
INSERT INTO products (product_name, category, price) VALUES
    ('Laptop', 'Electronics', 1200.00),
    ('Bookshelf', 'Furniture', 150.00),
    ('Headphones', 'Electronics', 50.00);

-- Query data using the secondary index
SELECT * FROM products WHERE category = 'Electronics';
```

Exercise 11.4: Dense Index

Objective: Create a table with a dense index, insert data, and observe the impact on query performance. A dense index is created on the "email" column of the "customers" table. Dense indexes are suitable for columns with a high cardinality, such as email addresses, where each value is distinct.

```
-- Create a table
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(100),
    phone_number VARCHAR(15)
);
```

```

-- Create a dense index on the email column
CREATE INDEX idx_email ON customers(email);

-- Insert sample data
INSERT INTO customers (first_name, last_name, email,
phone_number) VALUES
    ('John', 'Doe', 'john.doe@example.com', '555-1234'),
    ('Jane', 'Smith', 'jane.smith@example.com', '555-5678'),
    ('Bob', 'Johnson', 'bob.johnson@example.com', '555-9876');

-- Query data using the dense index
SELECT * FROM customers WHERE email =
'jane.smith@example.com';
-----

```

Exercise 11.5: Simulating Sparse Index

Creating a sparse index in PostgreSQL isn't a direct feature, as PostgreSQL doesn't provide a built-in "sparse index" type. However, we can simulate a sparse index scenario by considering a situation where indexing is selective based on a certain condition.

Objective: Create a table, add a conditional index (simulating a sparse index), insert data, and observe the impact on query performance. In this exercise, we're simulating a sparse index by creating a conditional index (idx_featured) that indexes only those rows where is_featured is true. This is a simplistic example, but in a real-world scenario, you might have situations where you want to index only a subset of your data.

```

-- Create a table
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10, 2),
    is_featured BOOLEAN
);

-- Insert sample data
INSERT INTO products (product_name, category, price,
is_featured) VALUES
    ('Laptop', 'Electronics', 1200.00, true),
    ('Bookshelf', 'Furniture', 150.00, false),
    ('Headphones', 'Electronics', 50.00, true);

-- Create a conditional index for featured products
CREATE INDEX idx_featured ON products(product_id) WHERE
is_featured = true;

-- Query data using the simulated sparse index
SELECT * FROM products WHERE is_featured = true;
-----

```

Exercise 11.6: Simulating Multilevel Index

Objective: Create a table, add a two-level index, insert data, and observe the impact on query performance. In this exercise, we've created a table of employees and established a two-level index structure. The first level is an index on the "department" column, and the second level is an index on the "salary" column within each department. This simulates a scenario where you might have a hierarchical index structure for efficient querying based on both department and salary.

```
-- Create a table
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    department VARCHAR(50),
    salary INTEGER
);

-- Insert sample data
INSERT INTO employees (first_name, last_name, department,
salary) VALUES
    ('John', 'Doe', 'IT', 80000),
    ('Jane', 'Smith', 'HR', 70000),
    ('Bob', 'Johnson', 'IT', 75000),
    ('Alice', 'Williams', 'Finance', 90000);

-- Create an index on department
CREATE INDEX idx_department ON employees(department);

-- Create a second-level index on salary within each
department
CREATE INDEX idx_salary ON employees(department, salary);

-- Query data using the multilevel index
SELECT * FROM employees WHERE department = 'IT' AND salary >
75000;
```

Exercise 11.7: B-tree Index in PostgreSQL

Objective: Create a table, add a B-tree index, insert data, and observe the impact on query performance. In this exercise, we're creating a table of products and adding a B-tree index on the "price" column. B-tree indexes are suitable for range queries, making them efficient for scenarios like querying products with prices greater than a certain value.

```
-- Create a table
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10, 2)
```

```

);

-- Insert sample data
INSERT INTO products (product_name, category, price) VALUES
    ('Laptop', 'Electronics', 1200.00),
    ('Bookshelf', 'Furniture', 150.00),
    ('Headphones', 'Electronics', 50.00);

-- Create a B-tree index on the price column
CREATE INDEX idx_price ON products(price);

-- Query data using the B-tree index
SELECT * FROM products WHERE price > 100.00;
-----

```

Exercise 11.8: Hash Index in PostgreSQL

Objective: Create a table, add a hash index, insert data, and observe the impact on query performance. In this exercise, we've created a table of students and added a hash index on the "age" column. Hash indexes are useful for equality queries, making them efficient for scenarios where you want to look up rows based on a specific value.

It's important to note that hash indexes in PostgreSQL are not suitable for range queries or ordering, so they are typically used in scenarios where equality checks are common.

```

-- Create a table
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    age INTEGER
);

-- Insert sample data
INSERT INTO students (first_name, last_name, age) VALUES
    ('John', 'Doe', 21),
    ('Jane', 'Smith', 22),
    ('Bob', 'Johnson', 20),
    ('Alice', 'Williams', 23);

-- Create a hash index on the age column
CREATE INDEX idx_age_hash ON students USING hash(age);

-- Query data using the hash index
SELECT * FROM students WHERE age = 22;

-----End-----

```