Laboratory Work 8: SQL Indexes

Objective

To understand and practice creating and managing different types of indexes in PostgreSQL, learning how indexes improve query performance and when to use them effectively.

Prerequisites

- Completion of Laboratory Work 7 (Views and Roles)
- Basic SQL knowledge (SELECT, INSERT, UPDATE, DELETE)
- Understanding of query performance concepts
- Access to a PostgreSQL database system

Theoretical Background

What are Indexes?

Indexes are database objects that improve the speed of data retrieval operations on tables. They work like an index in a book - instead of reading every page to find information, you can look up the index to quickly locate what you need.

Why Use Indexes?

- **Speed up queries**: Faster data retrieval, especially on large tables
- **Improve JOIN performance**: Speed up queries that join multiple tables
- **Enforce uniqueness**: Ensure unique values in columns
- **Sort data efficiently**: Help with ORDER BY operations

Index Types in PostgreSQL

1. **B-tree**: Default type, good for equality and range queries
2. **Hash**: Only for equality comparisons
3. **GiST**: For geometric and full-text search data
4. **GIN**: For array and full-text search operations

Important Considerations

- Indexes speed up reads but slow down writes (INSERT, UPDATE, DELETE)
- Don't create indexes on every column - only on frequently queried columns
- Indexes consume disk space
- PostgreSQL automatically creates indexes for PRIMARY KEY and UNIQUE constraints

Part 1: Database Setup

Use the same tables from Lab 6 and Lab 7 (employees, departments, projects). If you need to recreate them, here's a quick setup:

sql

```sql
-- Create tables
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50),
    location VARCHAR(50)
);

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT,
    salary DECIMAL(10,2),
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);

CREATE TABLE projects (
    proj_id INT PRIMARY KEY,
    proj_name VARCHAR(100),
    budget DECIMAL(12,2),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);

-- Insert sample data
INSERT INTO departments VALUES
(101, 'IT', 'Building A'),
(102, 'HR', 'Building B'),
(103, 'Operations', 'Building C');

INSERT INTO employees VALUES
(1, 'John Smith', 101, 50000),
(2, 'Jane Doe', 101, 55000),
(3, 'Mike Johnson', 102, 48000),
(4, 'Sarah Williams', 102, 52000),
(5, 'Tom Brown', 103, 60000);

INSERT INTO projects VALUES
(201, 'Website Redesign', 75000, 101),
(202, 'Database Migration', 120000, 101),
(203, 'HR System Upgrade', 50000, 102);
```

Part 2: Creating Basic Indexes

Exercise 2.1: Create a Simple B-tree Index

Create an index on the `salary` column of the `employees` table:

sql

```
CREATE INDEX emp_salary_idx ON employees(salary);
```
Verify the index was created:

sql
```
-- List all indexes on employees table
SELECT indexname, indexdef
FROM pg_indexes
WHERE tablename = 'employees';
```
Question: How many indexes exist on the employees table? (Hint: PRIMARY KEY creates an automatic index)

Exercise 2.2: Create an Index on a Foreign Key

Create an index on the `dept_id` column in the `employees` table:

sql
```
CREATE INDEX emp_dept_idx ON employees(dept_id);
```
Test the index usage:

sql
```
-- This query should use the index
SELECT * FROM employees WHERE dept_id = 101;
```
Question: Why is it beneficial to index foreign key columns?

Exercise 2.3: View Index Information

Use PostgreSQL system catalogs to view detailed index information:

sql
```
-- View all indexes in your database
SELECT
    tablename,
    indexname,
    indexdef
FROM pg_indexes
WHERE schemaname = 'public'
ORDER BY tablename, indexname;
```
Question: List all the indexes you see. Which ones were created automatically?

Part 3: Multicolumn Indexes

Exercise 3.1: Create a Multicolumn Index

Create an index on both `dept_id` and `salary` columns:

sql
```
CREATE INDEX emp_dept_salary_idx ON employees(dept_id, salary);
```
Test the multicolumn index:

sql
```
-- This query can use the multicolumn index
SELECT emp_name, salary
FROM employees
WHERE dept_id = 101 AND salary > 52000;
```
Question: Would this index be useful for a query that only filters by `salary` (without `dept_id`)? Why or why not?

Exercise 3.2: Understanding Column Order

Create another multicolumn index with reversed column order:

sql
```
CREATE INDEX emp_salary_dept_idx ON employees(salary, dept_id);
```
Compare with queries:

sql
```
-- Query 1: Filters by dept_id first
SELECT * FROM employees WHERE dept_id = 102 AND salary > 50000;

-- Query 2: Filters by salary first
SELECT * FROM employees WHERE salary > 50000 AND dept_id = 102;
```
Question: Does the order of columns in a multicolumn index matter? Explain.

Part 4: Unique Indexes

Exercise 4.1: Create a Unique Index

First, add a new column for employee email:

sql

```sql
ALTER TABLE employees ADD COLUMN email VARCHAR(100);

UPDATE employees SET email = 'john.smith@company.com' WHERE emp_id = 1;
UPDATE employees SET email = 'jane.doe@company.com' WHERE emp_id = 2;
UPDATE employees SET email = 'mike.johnson@company.com' WHERE emp_id = 3;
UPDATE employees SET email = 'sarah.williams@company.com' WHERE emp_id = 4;
UPDATE employees SET email = 'tom.brown@company.com' WHERE emp_id = 5;
```

Now create a unique index on the email column:

sql
```sql
CREATE UNIQUE INDEX emp_email_unique_idx ON employees(email);
```

Test the uniqueness constraint:

sql
```sql
-- This should fail with a unique violation error
INSERT INTO employees (emp_id, emp_name, dept_id, salary, email)
VALUES (6, 'New Employee', 101, 55000, 'john.smith@company.com');
```

Question: What error message did you receive?

Exercise 4.2: Unique Index vs UNIQUE Constraint

Check what indexes exist after adding a UNIQUE constraint:

sql
```sql
-- Add a phone column with UNIQUE constraint
ALTER TABLE employees ADD COLUMN phone VARCHAR(20) UNIQUE;
```

View the indexes:

sql
```sql
SELECT indexname, indexdef
FROM pg_indexes
WHERE tablename = 'employees' AND indexname LIKE '%phone%';
```

Question: Did PostgreSQL automatically create an index? What type of index?

Part 5: Indexes and Sorting

Exercise 5.1: Create an Index for Sorting

Create an index optimized for descending salary queries:

sql
```
CREATE INDEX emp_salary_desc_idx ON employees(salary DESC);
```
Test with an ORDER BY query:

sql
```
SELECT emp_name, salary
FROM employees
ORDER BY salary DESC;
```
Question: How does this index help with ORDER BY queries?

Exercise 5.2: Index with NULL Handling

Create an index that handles NULL values specially:

sql
```
CREATE INDEX proj_budget_nulls_first_idx ON projects(budget NULLS FIRST);
```
Test the index:

sql
```
SELECT proj_name, budget
FROM projects
ORDER BY budget NULLS FIRST;
```

Part 6: Indexes on Expressions

Exercise 6.1: Create a Function-Based Index

Create an index for case-insensitive employee name searches:

sql
```
CREATE INDEX emp_name_lower_idx ON employees(LOWER(emp_name));
```
Test the expression index:

sql
```
-- This query can use the expression index
SELECT * FROM employees WHERE LOWER(emp_name) = 'john smith';
```
Question: Without this index, how would PostgreSQL search for names case-insensitively?

Exercise 6.2: Index on Calculated Values

Add a hire_date column and create an index on the year:

sql
```sql
ALTER TABLE employees ADD COLUMN hire_date DATE;

UPDATE employees SET hire_date = '2020-01-15' WHERE emp_id = 1;
UPDATE employees SET hire_date = '2019-06-20' WHERE emp_id = 2;
UPDATE employees SET hire_date = '2021-03-10' WHERE emp_id = 3;
UPDATE employees SET hire_date = '2020-11-05' WHERE emp_id = 4;
UPDATE employees SET hire_date = '2018-08-25' WHERE emp_id = 5;

-- Create index on the year extracted from hire_date
CREATE INDEX emp_hire_year_idx ON employees(EXTRACT(YEAR FROM hire_date));
```
Test the index:

sql
```sql
SELECT emp_name, hire_date
FROM employees
WHERE EXTRACT(YEAR FROM hire_date) = 2020;
```

Part 7: Managing Indexes

Exercise 7.1: Rename an Index

Rename the `emp_salary_idx` index to `employees_salary_index`:

sql
```sql
ALTER INDEX emp_salary_idx RENAME TO employees_salary_index;
```
Verify the rename:

sql
```sql
SELECT indexname FROM pg_indexes WHERE tablename = 'employees';
```

Exercise 7.2: Drop Unused Indexes

Drop the redundant multicolumn index we created earlier:

sql
```sql
DROP INDEX emp_salary_dept_idx;
```
Question: Why might you want to drop an index?

Exercise 7.3: Reindex

Rebuild an index to optimize its structure:

sql
```
REINDEX INDEX employees_salary_index;
```
When is REINDEX useful?

- After bulk INSERT operations
- When index becomes bloated
- After significant data modifications

Part 8: Practical Scenarios

Exercise 8.1: Optimize a Slow Query

Consider this query that runs frequently:

sql
```
SELECT e.emp_name, e.salary, d.dept_name
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
WHERE e.salary > 50000
ORDER BY e.salary DESC;
```
Create indexes to optimize this query:

sql
```
-- Index for the WHERE clause
CREATE INDEX emp_salary_filter_idx ON employees(salary) WHERE salary > 50000;

-- Index for the JOIN
-- (already created: emp_dept_idx)

-- Index for ORDER BY
-- (already created: emp_salary_desc_idx)
```

Exercise 8.2: Partial Index

Create an index only for high-budget projects (budget > 80000):

sql
```
CREATE INDEX proj_high_budget_idx ON projects(budget)
WHERE budget > 80000;
```

Test the partial index:

sql
```
SELECT proj_name, budget
FROM projects
WHERE budget > 80000;
```
Question: What's the advantage of a partial index compared to a regular index?

Exercise 8.3: Analyze Index Usage

Use EXPLAIN to see if indexes are being used:

sql
```
EXPLAIN SELECT * FROM employees WHERE salary > 52000;
```
Question: Does the output show an "Index Scan" or a "Seq Scan" (Sequential Scan)? What does this tell you?

Part 9: Index Types Comparison

Exercise 9.1: Create a Hash Index

Create a hash index on department name:

sql
```
CREATE INDEX dept_name_hash_idx ON departments USING HASH (dept_name);
```
Test the hash index:

sql
```
SELECT * FROM departments WHERE dept_name = 'IT';
```
Question: When should you use a HASH index instead of a B-tree index?

Exercise 9.2: Compare Index Types

Create both B-tree and Hash indexes on the project name:

sql
```
-- B-tree index
CREATE INDEX proj_name_btree_idx ON projects(proj_name);
```

```sql
-- Hash index
CREATE INDEX proj_name_hash_idx ON projects USING HASH (proj_name);
```

Test with different queries:

sql

```sql
-- Equality search (both can be used)
SELECT * FROM projects WHERE proj_name = 'Website Redesign';

-- Range search (only B-tree can be used)
SELECT * FROM projects WHERE proj_name > 'Database';
```

Part 10: Cleanup and Best Practices

Exercise 10.1: Review All Indexes

List all indexes and their sizes:

sql

```sql
SELECT
    schemaname,
    tablename,
    indexname,
    pg_size_pretty(pg_relation_size(indexname::regclass)) as index_size
FROM pg_indexes
WHERE schemaname = 'public'
ORDER BY tablename, indexname;
```

Question: Which index is the largest? Why?

Exercise 10.2: Drop Unnecessary Indexes

Identify and drop indexes that are duplicates or rarely used:

sql

```sql
-- Drop the duplicate expression indexes
DROP INDEX IF EXISTS proj_name_hash_idx;

-- Keep only necessary indexes
```

Exercise 10.3: Document Your Indexes

Create a view that documents all custom indexes:

sql

```sql
CREATE VIEW index_documentation AS
SELECT
    tablename,
    indexname,
    indexdef,
    'Improves salary-based queries' as purpose
FROM pg_indexes
WHERE schemaname = 'public'
  AND indexname LIKE '%salary%';

SELECT * FROM index_documentation;
```

Summary Questions

1. **What is the default index type in PostgreSQL?**
2. **Name three scenarios where you should create an index:**
3. **Name two scenarios where you should NOT create an index:**
4. **What happens to indexes when you INSERT, UPDATE, or DELETE data?**
5. **How can you check if a query is using an index?**

Best Practices Checklist

- Index columns used frequently in WHERE clauses
- Index foreign key columns
- Index columns used in JOIN conditions
- Index columns used in ORDER BY
- Don't over-index (indexes have overhead)
- Consider multicolumn indexes for queries with multiple filters
- Use partial indexes for frequently queried subsets
- Regularly analyze and remove unused indexes
- Use EXPLAIN to verify index usage
- Consider expression indexes for computed values

Additional Challenges (Optional)

1. Create an index that would optimize finding all employees hired in a specific month
2. Create a composite unique index on dept_id and email in employees table
3. Use EXPLAIN ANALYZE to compare query performance with and without indexes
4. Create a covering index that includes all columns needed for a specific query

Submission Requirements:

- SQL script with all CREATE INDEX statements
- Answers to all questions
- Screenshots of EXPLAIN output for at least one query
- Brief reflection on when to use indexes

Good luck!