DTS207TC – Database Development and Design

Lab 10: Database Indexing

Instructor: Xiaowu Sun
School of AI and Advanced Computing (AIAC), XJTLU

November 2025

Contents

1 Lab Part 1: Concept of Indexing in Databases

1.1 Theoretical Background

An index in a database is similar to an address book or the table of contents of a book. Rather than scanning every row in a table, the database can use the index to quickly locate the rows that match a given condition.

When we create an index on a column, the DBMS builds a special data structure (usually some kind of sorted structure) which allows it to find rows based on values in that column much more efficiently.

1.1.1 Why Do We Need Indexes?

Without an index, the database must perform a **full table scan** to find matching data: it reads each row and checks whether the row satisfies the query condition. For large tables (e.g., millions of rows such as national ID databases or large payment platforms), this becomes very slow.

Indexes help in:

- **Speed**: Reduce I/O and CPU cost by avoiding full scans.
- **Efficiency**: Narrow down the search space to only relevant blocks or rows.

Real-life analogy: Imagine a huge book. If you need the chapter on "Database Indexes" and there is no table of contents, you would have to flip through every page. With a table of contents (index), you can immediately jump to the correct page range.

In this lab, we mainly discuss two important index types in PostgreSQL: B-tree index and BRIN (Block Range Index).

1.2 B-tree Index

The **B-tree index** is generally the default index type in PostgreSQL. It is effective for queries that involve `=`, `>`, `<`, `BETWEEN` and similar operators, and works well on numeric, text, and date/time values.

Data is organized in a **balanced tree structure**, enabling fast search, insertion, and deletion.

Example Syntax: Creating a simple B-tree index

```sql
CREATE INDEX idx_column_name
ON table_name (column_name);
```

Real-life analogy Think of a phone book, where names are stored in alphabetical order. You do not read every entry; you jump directly to the approximate region and then narrow down.

1.3 BRIN (Block Range Index)

A **BRIN index** (Block Range INdex) is highly space-efficient. Instead of storing information for each row, it stores summary information (such as minimum and maximum values) for *ranges of blocks*.

It is especially useful when:

- The table is very large.

- Data is naturally ordered or clustered (e.g., time-series data, sequential IDs).
- Queries are range-based (e.g., “all trades between time A and time B”).

BRIN is called a *sparse* index, while B-tree is a *dense* index.

Example Syntax: Creating a BRIN index

```
CREATE INDEX idx_brin
ON table_name
USING BRIN (timestamp_column);
```

Real-life analogy: High-frequency trading data Consider a financial services company storing high-frequency trading data.

- Each row contains: instrument ID, timestamp, price, etc.
- Data is continuously appended, ordered by timestamp.
- Queries often ask: “Find all trades for EURUSD between 09:00 and 10:00.”

Because rows are stored in roughly chronological order, BRIN can skip large ranges of blocks that obviously cannot contain the requested time range, while using very little index space.

2 Lab Part 2: Practical Experiment on Indexing

This part of the lab demonstrates how indexes influence query performance in PostgreSQL.

All SQL statements in this section should be executed in **pgAdmin 4** using the **Query Tool** against your chosen database.

2.1 Part 2.1: Insert 100,000 Rows and Run a Search Query

Step 1: Insert a large number of rows

In this step, we populate a table `market` with approximately 100,000 rows.

SQL: Create Market Table

```sql

-- Drop the table if it already exists
DROP TABLE IF EXISTS public.market;

-- Create a new table named "market"
CREATE TABLE public.market (
    id          bigserial PRIMARY KEY,
    -- Auto-incrementing primary key
    symbol      text,
    -- Stock symbol (e.g., 'AAPL', 'SYM123')
    price       numeric(10,2),
    -- Price with 2 decimal places
    volume      integer,
    -- Trading volume
    ts          timestamp
    -- Timestamp of the record
);
```

SQL: Insert 100,000 Random Rows into `market`

```sql

-- Insert 100,000 randomly generated rows into the "market" table
INSERT INTO public.market (symbol, price, volume, ts)
SELECT
    'SYM' || (random()*1000)::int,     -- Random symbol like SYM123
    round((random()*100)::numeric,2),
    -- Random price between 0-100
    (random()*10000)::int,
    -- Random volume between 0-10000
    timestamp '2025-05-01' + (random()*365 || ' days')::interval
    -- Add random days (1 year range)
FROM generate_series(1, 100000);
-- Generate 100,000 rows
```

SQL: View Rows from `market`

```sql

-- View the first 100 rows from the market table
SELECT * FROM public.market LIMIT 100;


-- View ALL rows from the market table
-- Warning: This may return 100,000+ rows
SELECT * FROM public.market;
```

Task:

- Write and Run the script (or adapt the template).
- After execution, confirm that the table contains about 100,000 rows.

Step 2: Run a search query without an index

Below is an example range query on the `timestamp` column:

SQL: Example query without index

```sql
SELECT *
FROM public.market
WHERE ts BETWEEN '2025-05-01 09:00:00'
                 AND '2025-05-01 10:00:00';
```

What to do

- Execute the query in pgAdmin.
- Observe the execution time shown in the *Messages* tab (usually in milliseconds).
- This is the baseline time *without* an index on `timestamp`.

2.2 Create Indexes and Compare Query Time

Now we create indexes and see how they affect performance.

B-tree index (default indexing method)

In PostgreSQL, when you run a standard `CREATE INDEX` on a column, the default index type is usually B-tree. The following SQL block demonstrates how to create **B-tree indexes** on different columns of the `public.market` table. B-tree is PostgreSQL's default indexing method and is efficient for:

- equality lookups (`=`),
- range queries (`>`, `<`, `BETWEEN`),
- ordering operations such as `ORDER BY`.

The listing includes:

1. A B-tree index on the `symbol` column — improves filtering by stock symbol.
2. A B-tree index on the `timestamp` column — speeds up time-based queries.

3. A composite B-tree index on both `symbol` and `timestamp` — useful when a query filters both columns simultaneously.

SQL: Creating B-tree Indexes on Different Columns

```sql

-- Create an index on the 'symbol' column
CREATE INDEX idx_symbol_btree
ON public.market(symbol);

-- Create an index on the 'timestamp' column
CREATE INDEX idx_timestamp_btree
ON public.market(timestamp);

-- Create a composite index on both 'symbol' and 'timestamp'
   columns
CREATE INDEX idx_symbol_timestamp_btree
ON public.market(symbol, timestamp);
```

In this example, we compare how PostgreSQL executes the same equality lookup query with and without a B-tree index. Without an index on the `symbol` column, PostgreSQL has no choice but to perform a full table scan, meaning it must examine every row in the `market` table. This becomes increasingly inefficient as the table grows larger.

After creating a B-tree index on `symbol`, the database can directly locate matching rows using an index scan. This significantly reduces the number of examined rows and leads to faster query execution, especially on large datasets.

Non-index Version (Full Table Scan)

```sql

-- Query executed WITHOUT an index on 'symbol'
-- PostgreSQL must scan every row in the table
SELECT *
FROM public.market
WHERE symbol = 'SYM123';
```

Index Version (Index Scan)

```sql

-- Create the index (safe version)
DROP INDEX IF EXISTS idx_symbol_btree;
CREATE INDEX idx_symbol_btree
ON public.market(symbol);

-- Query now uses an index scan instead of a full scan
SELECT *
FROM public.market
WHERE symbol = 'SYM123';
```

What to Do

- First, run the range query **without** creating an index on `ts` and observe the execution time.
- Create the B-tree index on `ts` using the provided SQL command.
- Re-run the same range query:

  ```
  ts BETWEEN '2025-05-01 09:00:00' AND '2025-05-01 10:00:00'
  ```

  and compare the execution time with the no-index case.

2.3 Create BRIN Indexes and Compare Query Time

Now we repeat the same experiment using a BRIN index. BRIN (Block Range Index) is designed for very large tables whose data is physically ordered, such as timestamp logs.

BRIN index (block-range indexing method)

BRIN indexes do not store every key. Instead, they store summary information (min/max) for ranges of table blocks. They are extremely space-efficient and fast for scanning naturally ordered data.

The listing below shows how to create a BRIN index on the `ts` column:

SQL: Creating a BRIN Index on `ts`

```sql

-- Create BRIN index on the timestamp column
DROP INDEX IF EXISTS idx_brin_ts;
CREATE INDEX idx_brin_ts
ON public.market
USING BRIN (ts);
```

In this example, we compare how PostgreSQL executes the same timestamp range query with and without a BRIN index. Without an index, PostgreSQL performs a full table scan. With a BRIN index, PostgreSQL can skip large portions of irrelevant blocks, making range queries significantly faster on large, time-ordered data.

Non-index Version (Full Table Scan)

```sql

-- Query executed WITHOUT any index on 'ts'
-- PostgreSQL must scan the entire table
SELECT *
FROM public.market
WHERE ts BETWEEN '2025-05-01 09:00:00'
             AND '2025-05-01 10:00:00';
```

Index Version (BRIN Range Scan)

```sql

-- Create the BRIN index (safe version)
DROP INDEX IF EXISTS idx_brin_ts;
CREATE INDEX idx_brin_ts
ON public.market
USING BRIN (ts);

-- Query now benefits from the BRIN index
SELECT *
FROM public.market
WHERE ts BETWEEN '2025-05-01 09:00:00'
            AND '2025-05-01 10:00:00';
```

What to Do

- Run the timestamp range query without any index and record the execution time.
- Create the BRIN index on `ts` using the SQL code shown above.
- Re-run the same range query and compare the execution time.
- Compare the performance difference between the BRIN version (Part 3) and the B-tree version (Part 2).

3 How PostgreSQL Chooses an Index to apply?

3.1 Automatic Index Selection

When a query is executed, PostgreSQL does *not* blindly use all available indexes. Instead, the planner evaluates several factors to determine the most efficient plan:

- **Table size** — large tables benefit more from indexing.
- **Data distribution and ordering** — BRIN is effective when rows are physically ordered.
- **Query predicates** — equality vs. range queries may favor different index types.
- **Statistics** — PostgreSQL uses collected statistics to estimate selectivity.

The planner may choose:

- a sequential scan,
- a B-tree index scan,
- or a BRIN range scan.

3.2 Using EXPLAIN and EXPLAIN ANALYZE

Step 1: Creating Indexes

Before running the range queries, we first create two different types of indexes on the `ts` column of the `public.market` table. These indexes will later allow us to compare how PostgreSQL optimizes queries using either a **B-tree index** or a **BRIN index**.

The B-tree index is suitable for general-purpose lookups, while the BRIN index is designed for very large tables where values are naturally ordered (such as timestamps). The following SQL block prepares both indexes.

SQL: Creating Both B-tree and BRIN Indexes on `ts`

```

-- Remove existing indexes if they already exist
DROP INDEX IF EXISTS idx_btree_timestamp;
DROP INDEX IF EXISTS idx_brin_timestamp;

-- Create a B-tree index (default indexing method)
CREATE INDEX idx_btree_timestamp
ON public.market (ts);

-- Create a BRIN index (block-range index, good for large ordered
    data)
CREATE INDEX idx_brin_timestamp
ON public.market
USING BRIN (ts);
```

Step 2: Running Queries

Now that both indexes have been created, we can run queries on the `public.market` table that filter on the `ts` column. PostgreSQL will automatically choose the most appropriate index (B-tree or BRIN) depending on the characteristics of the query and the distribution of the underlying data.

In this step, you will execute a range query on the timestamp column and observe the execution time. This will later help you compare the performance difference between the two indexing methods.

The `EXPLAIN` command shows the planned execution method. The `EXPLAIN ANALYZE` command executes the query and provides measured timings.

Example Query1: using range query

```
EXPLAIN ANALYZE
SELECT *
FROM public.market
WHERE ts BETWEEN '2025-05-01 09:00:00'
             AND '2025-05-11 10:00:00';
```

Example Query2: using exact match query

```
EXPLAIN ANALYZE
SELECT * FROM public.market
WHERE ts = '2025-05-01 09:00:00';
```

3.3 What to do

Tasks for Students

- Run several timestamp range queries using different time windows.
- Use `EXPLAIN ANALYZE` to check whether PostgreSQL chooses the BRIN index, the B-tree index, or a sequential scan.

4 Summary

In this lab, you have:

- Understood the conceptual role of **indexes** in a database.
- Learned the difference between:
  - **B-tree** indexes (dense, per-row; good for exact match and general range queries).
  - **BRIN** indexes (sparse, per-block; good for large, naturally ordered data).
- Practically:
  - Inserted a large dataset (100,000 rows) into a table.
  - Measured query time without any index.
  - Created B-tree and BRIN indexes and compared performance.
  - Used `EXPLAIN ANALYZE` to see which index PostgreSQL chose.

These concepts and techniques form the foundation for more advanced topics such as query optimization, index-only scans, composite indexes, and cost-based planning in modern relational database systems.