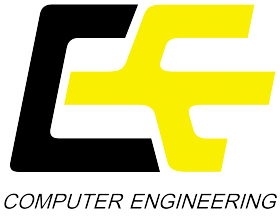

Modul Praktikum
Sistem Manajemen Basis Data

ACID dan High Concurrency PostgreSQL

Departemen Teknik Komputer
Institut Teknologi Sepuluh Nopember
Surabaya

Penulis: Arta Kusuma Hernanda

2025

Kata Pengantar

Puji syukur kami panjatkan kepada Tuhan Yang Maha Esa atas segala rahmat dan karunia-Nya sehingga Modul Praktikum Basis Data: ACID dan High Concurrency dengan PostgreSQL ini dapat tersusun hingga selesai. Modul ini disusun untuk memenuhi kebutuhan mahasiswa dalam pembelajaran mata kuliah Basis Data di Departemen Teknik Komputer, Institut Teknologi Sepuluh Nopember Surabaya.

Modul praktikum ini bertujuan untuk memberikan pemahaman mendalam tentang konsep ACID (Atomicity, Consistency, Isolation, Durability) dan High Concurrency pada database PostgreSQL melalui empat skenario praktis. Melalui modul ini, mahasiswa diharapkan dapat memahami dan mempraktikkan berbagai teknik optimasi database seperti indexing, transactions, partitioning, serta strategi backup dan recovery.

Kami menyadari bahwa modul praktikum ini masih jauh dari kesempurnaan. Oleh karena itu, kritik dan saran yang membangun sangat kami harapkan demi perbaikan modul ini di masa yang akan datang.

Akhir kata, kami berharap modul praktikum ini dapat bermanfaat bagi mahasiswa dalam mengembangkan kemampuan dalam pengelolaan basis data dan dapat menjadi bekal yang berguna di dunia kerja nantinya.

Surabaya, April 2025
Tim Penyusun

Daftar Isi

Persiapan Praktikum

1 Deskripsi Praktikum

Praktikum ini bertujuan untuk memahami konsep ACID (Atomicity, Consistency, Isolation, Durability) dan High Concurrency pada database PostgreSQL melalui empat skenario praktis.

Praktikum ini terdiri dari empat modul yang saling berkaitan:

1. **Indexing**: Mempelajari cara mengoptimalkan performa query dengan index dan mengukur dampaknya pada konkurensi tinggi.
2. **Transaction**: Mempelajari konsep ACID, tingkat isolasi transaksi, dan menangani konkurensi tinggi dengan benar.
3. **Partitioning**: Mempelajari cara membagi tabel besar menjadi bagian yang lebih kecil dan dampaknya terhadap performa.
4. **Backup dan Recovery**: Mempelajari strategi backup dan recovery untuk menjamin ketersediaan dan integritas data.

2 Persiapan Environment

2.1 Menggunakan Docker

1. Install Docker dan Docker Compose di komputer Anda
2. Unduh file praktikum yang sudah disediakan oleh asisten praktikum
3. Jalankan docker-compose file untuk membuat container dengan perintah:

```
docker-compose up -d
```

4. Generate data dummy:

```
cd scripts
pip install faker
python generate_data.py
```

5. Import data ke PostgreSQL:

```
cd scripts
bash import_data.sh
```

6. Akses pgAdmin di browser:
   - URL: http://localhost:5050
   - Email: admin@example.com
   - Password: p4ssw0rd
7. Tambahkan server di pgAdmin:
   - Name: PostgreSQL Praktikum
   - Host: postgres
   - Port: 5432
   - Username: praktikan
   - Password: p4ssw0rd

3 Petunjuk Penggunaan Modul

Setiap modul praktikum memiliki:

- PDF Modul
- File README.md dengan penjelasan konsep dan langkah-langkah
- File SQL dengan query untuk latihan
- Tugas

4 Panduan Pengerjaan

1. Praktikum dikerjakan dalam kelompok
2. Setiap kelompok mengerjakan semua modul praktikum
3. Pelaksanaan praktikum dilakukan oleh asisten praktikum dengan pengawasan koordinator dosen praktikum
4. Setiap kelompok membuat laporan akhir berisi:
   - Jawaban tugas dari setiap modul
   - Analisis dan kesimpulan

5 Struktur Folder Praktikum

Folder praktikum memiliki struktur sebagai berikut:

```
postgres-praktikum/
|-- docker-compose.yml
|-- init/
|   |-- 01-schema.sql
|   `-- 02-functions.sql
|-- data/
|   |-- products.csv
|   |-- customers.csv
|   |-- orders.csv
|   `-- order_items.csv
|-- scripts/
|   |-- generate_data.py
|   |-- import_data.sh
|   |-- benchmark.sh
|   `-- README.md
`-- praktikum/
|   |-- praktikum1_indexing/
|   |-- praktikum2_transaction/
|   |-- praktikum3_partitioning/
`-- |-- praktikum4_backup/
```

6 Dataset Praktikum

Dataset yang digunakan adalah data e-commerce dummy yang berisi:

- 10.000 produk
- 5.000 pelanggan
- 20.000 pesanan
- 50.000 item pesanan

7 Referensi

- PostgreSQL Documentation
- PostgreSQL Concurrency
- PostgreSQL Indexing
- PostgreSQL Partitioning
- PostgreSQL Backup and Restore

Bab 1

Implementasi Indexing pada PostgreSQL

1 Pendahuluan

Index pada database adalah struktur data khusus yang mempercepat operasi pencarian dan pengambilan data. Analoginya seperti indeks pada buku yang membantu kita menemukan konten tertentu dengan cepat tanpa perlu membaca seluruh buku. Dalam PostgreSQL, index disimpan dalam struktur khusus yang mengoptimalkan pencarian berdasarkan kolom tertentu.

Implementasi indexing yang tepat merupakan salah satu strategi utama untuk meningkatkan performa database, terutama pada sistem dengan data besar dan tingkat konkurensi tinggi.

1.1 Tujuan Praktikum

Dilaksanakannya praktikum indexing, praktikan diharapkan mampu:

1. Memahami konsep dasar indexing pada database PostgreSQL
2. Mempelajari tipe-tipe index di PostgreSQL
3. Mengimplementasikan index yang tepat untuk sebuah skenario
4. Menganalisis dampak index terhadap performa query konkuren

2 Konsep Dasar Indexing

2.1 Apa itu Index?

Index pada database bekerja mirip dengan indeks buku: membantu sistem menemukan data dengan cepat tanpa memeriksa seluruh tabel (table scan). Index menyimpan pointer ke baris data dalam tabel berdasarkan nilai kolom yang diindeks.

2.2 Bagaimana Index Bekerja

Ketika query dijalankan, PostgreSQL menganalisis apakah menggunakan index akan lebih efisien daripada melakukan table scan. Jika menggunakan index lebih efisien, PostgreSQL akan:

1. Mencari data di index berdasarkan kondisi WHERE
2. Menemukan referensi (pointer) ke baris data yang relevan
3. Mengambil data dari tabel menggunakan pointer tersebut

2.3 Kapan Menggunakan Index

Index sangat berguna pada:

1. Kolom yang sering digunakan dalam klausa WHERE
2. Kolom yang sering digunakan untuk JOIN antar tabel
3. Kolom yang sering digunakan dalam klausa ORDER BY atau GROUP BY
4. Tabel berukuran besar dengan query yang hanya mengambil sebagian kecil data

2.4 Kapan Tidak Menggunakan Index

Index tidak selalu bermanfaat pada:

1. Tabel kecil yang lebih efisien dilakukan sequential scan
2. Kolom dengan kardinalitas rendah (nilai yang berbeda sedikit)
3. Kolom yang jarang digunakan dalam query
4. Tabel yang sering diupdate/dihapus (overhead pemeliharaan index)

2.5 Dampak Index pada Performa

Index memberikan dampak pada:

1. **SELECT**: Mempercepat query dengan mengurangi jumlah baris yang perlu diperiksa
2. **INSERT**: Memerlukan overhead tambahan untuk memperbarui index
3. **UPDATE**: Memerlukan overhead untuk memperbarui index jika kolom yang diindex berubah
4. **DELETE**: Memerlukan overhead untuk memperbarui index

3 Jenis-jenis Index di PostgreSQL

PostgreSQL menyediakan beberapa jenis index yang dapat dipilih sesuai dengan kebutuhan:

3.1 B-tree Index

1. Index default di PostgreSQL
2. Efisien untuk perbandingan dengan operator equality (=) dan range ($<, >, <=, >=, BETWEEN$)
3. Mendukung urutan untuk ORDER BY
4. Cocok untuk kebanyakan kasus penggunaan

3.2 Hash Index

1. Optimal hanya untuk operator equality (=)
2. Lebih cepat dari B-tree untuk operasi equality sederhana
3. Tidak mendukung range queries atau sorting
4. Dirancang untuk tabel hash di memori

3.3 GiST Index (Generalized Search Tree)

1. Index untuk data geometri, text-search, dan data kompleks lainnya
2. Fleksibel, dapat digunakan untuk data custom
3. Mendukung nearest-neighbor searches
4. Digunakan untuk full-text search, data geografis, dll

3.4 GIN Index (Generalized Inverted Index)

1. Dirancang untuk nilai yang memiliki multiple components (arrays, jsonb, text-search)
2. Efisien untuk pencarian yang membutuhkan matching multiple values
3. Cocok untuk kolom yang menyimpan data semi-structured
4. Lebih lambat untuk operasi insert dibanding GiST

3.5 BRIN Index (Block Range INdex)

1. Dirancang untuk tabel sangat besar dengan data yang terurut secara natural
2. Sangat kecil dan efisien untuk kolom seperti timestamp, ID sequential, dll
3. Kinerja query lebih rendah dari B-tree tapi overhead penyimpanan jauh lebih kecil

3.6 SP-GiST Index (Space-Partitioned GiST)

1. Untuk data yang bisa dipartisi secara non-overlapping
2. Baik untuk data hierarchical seperti rentang IP, geo-data
3. Mendukung nearest-neighbor searches

Jenis Index	Karakteristik dan Penggunaan
B-tree	Index default PostgreSQL. Efisien untuk operasi perbandingan (=, ¡, ¿, BETWEEN) dan mendukung pengurutan (ORDER BY).
Hash	Khusus untuk operasi equality (=). Lebih cepat dari B-tree untuk lookup sederhana, tidak mendukung range.
GiST/GIN	Untuk data kompleks seperti full-text search, data spasial, array, dan JSON. GIN lebih lambat untuk insert tapi lebih cepat untuk search.
BRIN	Untuk tabel besar dengan data terurut secara natural (seperti timestamp). Ukuran kecil, overhead rendah.

Gambar 1.1: Jenis-jenis Index di PostgreSQL dan Penggunaannya

3.7 Benchmark pada Konkurensi Tinggi

Pada situasi konkurensi tinggi, performa index dapat dipengaruhi oleh beberapa faktor:

- **Contention**: Kompetisi antar koneksi untuk mengakses data yang sama
- **Lock contention**: Waktu tunggu karena lock pada baris atau tabel
- **Index bloat**: Overhead karena index yang jarang di-maintenance

4 Best Practices Implementasi Indexing

1. **Index kolom yang sering digunakan dalam WHERE, JOIN, ORDER BY**
2. **Hindari over-indexing**: Terlalu banyak index berdampak negatif pada performa INSERT/UPDATE/DELETE
3. **Gunakan composite index** untuk query dengan multiple conditions
4. **Perhatikan urutan kolom** pada composite index (most selective first)
5. **Pertimbangkan partial index** untuk subset data yang sering diakses
6. **Gunakan EXPLAIN ANALYZE** untuk validasi penggunaan index

7. **Jalankan VACUUM ANALYZE** secara berkala untuk memperbarui statistik
8. **Monitor ukuran dan penggunaan index** menggunakan pg_stat_*
9. **Lakukan REINDEX** pada index yang terfragmentasi
10. **Evaluasi trade-off** antara kecepatan query vs overhead pemeliharaan

5 Tahapan Praktikum

5.1 Persiapan

1. Pastikan container Docker PostgreSQL sudah berjalan
2. Gunakan pgAdmin atau PSQL untuk mengakses database
3. Pastikan data sudah diimpor menggunakan script yang disediakan

5.2 Mengeksplorasi Execution Plan

Pertama, Anda akan mempelajari cara melihat execution plan query menggunakan EXPLAIN dan EXPLAIN ANALYZE:

```
-- Menampilkan execution plan
EXPLAIN SELECT * FROM products WHERE category = '
 Elektronik';

-- Menampilkan execution plan beserta runtime
EXPLAIN ANALYZE SELECT * FROM products WHERE category = '
 Elektronik';
```

5.3 Mengukur Performa Query Tanpa Index

1. Jalankan query berikut tanpa index dan catat waktu eksekusinya:

```
    -- Query 1: Filter berdasarkan kategori
    SELECT COUNT(*) FROM products WHERE category = '
   Elektronik';

    -- Query 2: Filter berdasarkan range harga
    SELECT * FROM products WHERE price BETWEEN 1000000
    AND 5000000;

```

```
    -- Query 3: Filter berdasarkan tanggal pesanan
    SELECT o.order_id, o.customer_id, o.order_date
    FROM orders o
    WHERE o.order_date BETWEEN '2023-06-01' AND '
   2023-06-30';

    -- Query 4: Join tanpa index
    SELECT c.first_name, c.last_name, o.order_id, o.
   order_date, o.total_amount
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    WHERE c.city = 'Jakarta';

    -- Query 5: Aggregate yang berat
    SELECT p.category, COUNT(*) as total_products, AVG
   (p.price) as avg_price
    FROM products p
    GROUP BY p.category
    ORDER BY total_products DESC;

```

2. Lakukan benchmark konkuren tanpa index:

```
    # Di terminal
    bash query_before.sql 20 100 10

```

5.4 Implementasi Index yang Tepat

Sekarang, tambahkan index yang sesuai untuk setiap query:

```
-- Hapus index yang sudah ada (untuk tujuan praktikum)
DROP INDEX IF EXISTS idx_products_category;
DROP INDEX IF EXISTS idx_products_price;
DROP INDEX IF EXISTS idx_orders_date;
DROP INDEX IF EXISTS idx_customers_city;

-- Index 1: B-Tree index untuk kategori produk
CREATE INDEX idx_products_category ON products(category);

-- Index 2: B-Tree index untuk range harga
CREATE INDEX idx_products_price ON products(price);

-- Index 3: B-Tree index untuk tanggal pesanan
```

```
CREATE INDEX idx_orders_date ON orders(order_date);

-- Index 4: Index untuk city pada tabel customers
CREATE INDEX idx_customers_city ON customers(city);

-- Index 5: Composite index untuk JOIN operation
CREATE INDEX idx_orders_customer_id ON orders(customer_id
 );
```

5.5 Mengukur Performa Setelah Indexing

1. Jalankan kembali query yang sama dan bandingkan waktu eksekusinya:

```
    -- Jalankan semua query yang sama seperti
   sebelumnya dan bandingkan execution plan
    EXPLAIN ANALYZE SELECT COUNT(*) FROM products
   WHERE category = 'Elektronik';
    -- dan seterusnya untuk query lainnya

```

2. Lakukan benchmark konkuren dengan index:

```
    # Di terminal
    cd scripts
    bash benchmark.sh ../praktikum/praktikum1_indexing
   /query_after.sql 20 100 10

```

5.6 Mempelajari Tipe Index Lain

PostgreSQL mendukung beberapa tipe index. Cobalah implementasi berikut:

```
-- Index Partial untuk produk dengan stok rendah
CREATE INDEX idx_products_low_stock ON products(
 product_id, stock_quantity)
WHERE stock_quantity < 10;

-- Index menggunakan ekspresi untuk pencarian case-
 insensitive
CREATE INDEX idx_products_name_lower ON products(LOWER(
 name));

-- BRIN Index untuk data berurutan (seperti timestamp)
```

```
9  CREATE INDEX idx_orders_date_brin ON orders USING BRIN(
    order_date);
10
11 -- GIN Index untuk pencarian full-text (jika menggunakan
    PostgreSQL >= 9.6)
12 CREATE INDEX idx_products_description_gin ON products
13 USING GIN(to_tsvector('english', description));
```

6 Referensi

- PostgreSQL Documentation: Indexes
- PostgreSQL Documentation: EXPLAIN
- PostgreSQL Documentation: Performance Tips
- PostgreSQL Documentation: Index Types
- PostgreSQL Documentation: pgbench

Bab 2

Implementasi Transaction dan ACID pada PostgreSQL

1 Pendahuluan

Transaksi (transaction) merupakan salah satu konsep fundamental dalam sistem basis data relasional yang menjamin integritas dan reliabilitas data. Dalam PostgreSQL, implementasi transaksi yang tepat dan pemahaman ACID properties sangat penting untuk mengembangkan aplikasi yang handal, terutama dalam lingkungan dengan konkurensi tinggi.

Transaksi adalah unit kerja yang mengubah data dari satu state ke state lainnya. Semua perubahan dalam transaksi harus berhasil secara lengkap, atau tidak ada yang dilakukan sama sekali (all-or-nothing). Konsep ini menjamin bahwa database tetap dalam keadaan konsisten meskipun terjadi kegagalan sistem.

1.1 Tujuan Praktikum

Dilaksanakannya praktikum transaction dan ACID, praktikan diharapkan mampu:

1. Memahami konsep ACID dalam transaksi database
2. Mengimplementasikan transaksi pada PostgreSQL
3. Mempelajari tingkat isolasi transaksi
4. Menganalisis fenomena konkurensi seperti dirty read, non-repeatable read, dan phantom read
5. Mengimplementasikan strategi penanganan konkurensi tinggi

2 Konsep ACID dalam Transaksi Database

ACID adalah akronim yang merepresentasikan empat properti penting dari transaksi database yang menjamin reliabilitas data meskipun terjadi kegagalan sistem, error, atau konkurensi akses.

2.1 Atomicity (Atomisitas)

1. Transaksi harus dijalankan sepenuhnya atau dibatalkan sepenuhnya
2. Tidak ada hasil parsial; jika satu operasi gagal, semua operasi dibatalkan
3. Implementasi menggunakan mekanisme COMMIT dan ROLLBACK
4. Contoh: Transfer uang antar rekening harus mengurangi saldo satu rekening dan menambah rekening lain, atau tidak melakukan perubahan sama sekali

2.2 Consistency (Konsistensi)

1. Transaksi harus membawa database dari satu state valid ke state valid lainnya
2. Semua aturan integritas data (constraints, cascade, triggers) harus tetap terpenuhi
3. Jumlah total saldo sebelum dan sesudah transaksi harus tetap sama
4. Melindungi aplikasi dari data yang tidak konsisten

2.3 Isolation (Isolasi)

1. Transaksi harus berjalan seolah-olah tidak ada transaksi lain yang berjalan secara bersamaan
2. Mencegah transaksi membaca data ”kotor” dari transaksi lain yang belum commit
3. PostgreSQL mengimplementasikan multi-version concurrency control (MVCC)
4. Terdapat beberapa tingkat isolasi yang bisa dipilih sesuai kebutuhan

2.4 Durability (Durabilitas)

1. Perubahan yang sudah di-commit harus tetap tersimpan, bahkan jika terjadi kegagalan sistem
2. PostgreSQL menggunakan Write-Ahead Logging (WAL) untuk menjamin durabilitas
3. Data yang sudah di-commit harus bertahan dari crash, power failure, dll
4. Memungkinkan pemulihan data setelah kegagalan sistem

3 Tingkat Isolasi Transaksi

PostgreSQL mendukung beberapa tingkat isolasi sesuai dengan standar SQL. Setiap tingkat isolasi menawarkan keseimbangan antara konsistensi dan performa.

3.1 READ UNCOMMITTED

1. PostgreSQL tidak benar-benar mengimplementasikan tingkat ini
2. Dalam PostgreSQL, READ UNCOMMITTED sama dengan READ COMMITTED
3. Tidak mungkin membaca perubahan yang belum di-commit (dirty read)

3.2 READ COMMITTED (Default)

1. Tingkat isolasi default di PostgreSQL
2. Transaksi hanya dapat membaca data yang sudah di-commit
3. Mencegah dirty read, tetapi memungkinkan non-repeatable read dan phantom read
4. Membaca yang sama dalam satu transaksi dapat mengembalikan hasil berbeda jika transaksi lain melakukan commit

3.3 REPEATABLE READ

1. Transaksi hanya melihat data yang sudah di-commit sebelum transaksi dimulai
2. Mencegah dirty read dan non-repeatable read
3. Melindungi dari perubahan data yang dibaca, tetapi masih memungkinkan phantom read
4. Semua query dalam satu transaksi melihat snapshot database yang sama

3.4 SERIALIZABLE

1. Tingkat isolasi tertinggi
2. Mencegah semua anomali konkurensi (dirty read, non-repeatable read, phantom read)
3. Transaksi berjalan seolah-olah dieksekusi secara serial (satu per satu)
4. Dapat menyebabkan serialization error yang memerlukan retry
5. Memberikan konsistensi tertinggi dengan trade-off performa

Tingkat Isolasi	**Dirty Read**	**Non-repeatable Read**	**Phantom Read**
READ UNCOMMITTED*	Tidak	Ya	Ya
READ COMMITTED	Tidak	Ya	Ya
REPEATABLE READ	Tidak	Tidak	Ya**
SERIALIZABLE	Tidak	Tidak	Tidak

Gambar 2.1: Tingkat Isolasi dan Anomali Konkurensi di PostgreSQL
Sama dengan READ COMMITTED di PostgreSQL
*Di PostgreSQL, REPEATABLE READ juga mencegah phantom read, yang melebihi standar SQL

3.5 Anomali Konkurensi

Berikut adalah penjelasan lebih detail mengenai anomali yang dapat terjadi:

1. **Dirty Read**: Membaca data yang belum di-commit oleh transaksi lain

2. **Non-repeatable Read**: Transaksi membaca data dua kali, tetapi antara dua pembacaan tersebut, data diubah dan di-commit oleh transaksi lain

3. **Phantom Read**: Transaksi mengeksekusi query yang mengembalikan himpunan baris, tetapi transaksi lain insert/delete baris yang memenuhi kondisi query

4 Strategi Penanganan Konkurensi

Untuk menangani konkurensi dengan baik, PostgreSQL menyediakan beberapa strategi:

4.1 Pessimistic Concurrency Control

1. Menggunakan locking untuk mencegah konflik

2. Row-level locking dengan FOR UPDATE, FOR SHARE, dll

3. Table-level locking dengan LOCK TABLE

4. Mencegah konflik sebelum terjadi

5. Dapat menyebabkan deadlock

4.2 Optimistic Concurrency Control

1. Mengasumsikan konflik jarang terjadi

2. Tidak mengunci data, tetapi memeriksa apakah data berubah sebelum commit

3. Biasanya menggunakan version number atau timestamp

4. Cocok untuk lingkungan dengan read-heavy workload

5. Memerlukan retry jika terjadi konflik

4.3 Multi-Version Concurrency Control (MVCC)

1. PostgreSQL menggunakan MVCC untuk implementasi isolasi
2. Setiap transaksi melihat snapshot database pada titik waktu tertentu
3. Memungkinkan reader tidak mengunci writer dan sebaliknya
4. Meningkatkan konkurensi karena read tidak menghalangi write
5. Trade-off adalah overhead penyimpanan untuk multiple versions

5 Tahapan Praktikum

5.1 Persiapan

1. Pastikan container Docker PostgreSQL sudah berjalan
2. Gunakan pgAdmin atau PSQL untuk mengakses database
3. Pastikan data sudah diimpor menggunakan script yang disediakan

5.2 Implementasi Transaksi Dasar

Mari kita mulai dengan mempelajari cara mengimplementasikan transaksi dasar di PostgreSQL:

```
1  -- Transaksi dasar
2  BEGIN;
3  UPDATE products SET stock_quantity = stock_quantity - 5
    WHERE product_id = 1;
4  INSERT INTO orders (customer_id, status) VALUES (1, '
    pending');
5  COMMIT;
6
7  -- Transaksi dengan ROLLBACK
8  BEGIN;
9  UPDATE products SET stock_quantity = stock_quantity - 100
     WHERE product_id = 1;
10 -- Oops, stok tidak cukup!
11 ROLLBACK;
```

5.3 Demonstrasi Tingkat Isolasi Transaksi

Selanjutnya, kita akan mendemonstrasikan perbedaan antara berbagai tingkat isolasi:

READ COMMITTED (Default)

Buka dua terminal PostgreSQL secara bersamaan dan jalankan:
Terminal 1:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE bank_accounts SET balance = balance - 1000000
 WHERE account_id = 1;
-- Tunggu beberapa saat sebelum commit
SELECT pg_sleep(30);
COMMIT;
```

Terminal 2:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- Ini akan menampilkan nilai sebelum update di Terminal
 1
SELECT * FROM bank_accounts WHERE account_id = 1;
-- Tunggu Terminal 1 commit
SELECT pg_sleep(35);
-- Ini akan menampilkan nilai setelah update di Terminal
 1
SELECT * FROM bank_accounts WHERE account_id = 1;
COMMIT;
```

REPEATABLE READ

Buka dua terminal PostgreSQL dan jalankan:
Terminal 1:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM bank_accounts WHERE account_id = 1;
-- Tunggu Terminal 2 melakukan update dan commit
SELECT pg_sleep(30);
-- Ini masih akan menampilkan nilai asli meskipun
 Terminal 2 sudah commit
SELECT * FROM bank_accounts WHERE account_id = 1;
COMMIT;
```

Terminal 2:

```
BEGIN;
UPDATE bank_accounts SET balance = balance - 1000000
 WHERE account_id = 1;
COMMIT;
```

SERIALIZABLE

Buka dua terminal PostgreSQL dan jalankan:
Terminal 1:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT SUM(balance) FROM bank_accounts WHERE customer_id
 = 1;
-- Tunggu Terminal 2 menambahkan account baru
SELECT pg_sleep(30);
-- Ini tidak akan melihat account baru karena isolasi
 serializable
SELECT SUM(balance) FROM bank_accounts WHERE customer_id
 = 1;
COMMIT;
```

Terminal 2:

```
BEGIN;
INSERT INTO bank_accounts (customer_id, balance)
VALUES (1, 5000000);
COMMIT;
```

5.4 Mengatasi Kondisi Konkurensi Tinggi

Menggunakan Lock (Pessimistic Concurrency Control)

```
-- Row-level lock
BEGIN;
SELECT * FROM products WHERE product_id = 1 FOR UPDATE;
-- Sekarang row ini terkunci hingga transaksi selesai
UPDATE products SET stock_quantity = stock_quantity - 5
 WHERE product_id = 1;
COMMIT;

-- Lock pada tingkat tabel
BEGIN;
```

```
LOCK TABLE products IN EXCLUSIVE MODE;
-- Operasi pada tabel products
COMMIT;
```

Optimistic Concurrency Control

```
-- Menggunakan last_updated sebagai version control
BEGIN;
SELECT product_id, stock_quantity, last_updated
FROM products WHERE product_id = 1;

-- Asumsikan kita mendapatkan last_updated = '2023-07-15
 10:00:00'
UPDATE products
SET stock_quantity = stock_quantity - 5,
    last_updated = CURRENT_TIMESTAMP
WHERE product_id = 1 AND last_updated = '2023-07-15
 10:00:00';

-- Jika tidak ada baris yang terupdate, berarti data
 sudah berubah
-- dan kita harus mengulang transaksi
COMMIT;
```

5.5 Implementasi Function Transfer Uang

```
-- Function sudah dibuat di 02-functions.sql
-- Cobalah dari dua terminal berbeda
BEGIN;
SELECT * FROM bank_accounts WHERE account_id IN (1, 2);
SELECT transfer_money(1, 2, 500000);
SELECT * FROM bank_accounts WHERE account_id IN (1, 2);
COMMIT;
```

5.6 Pengujian Konkurensi Tinggi

Untuk menguji bagaimana transaksi berperilaku pada konkurensi tinggi, gunakan script benchmark:

```
# Di terminal
cd scripts
```

```
bash benchmark.sh ../praktikum/praktikum2_transaction/
 transfer_money.sql 20 100 10
```

5.7 Troubleshooting

Jika mengalami error duplicate key, cek sequence id pada tabel tersebut kemudian reset ke max dari id tabel:

```
-- tableName_columnName_seq
SELECT currval('orders_order_id_seq');
-- untuk reset
SELECT setval('orders_order_id_seq', (SELECT MAX(order_id
 ) FROM orders));
```

6 Tugas Praktikum

1. Implementasikan sistem pemesanan produk dengan transaksi yang:
   - Mengurangi stok produk
   - Membuat pesanan baru
   - Menambahkan item pesanan
   - Memastikan stok mencukupi
   - Harus ACID-compliant
2. Implementasikan dua strategi konkurensi berbeda dan bandingkan performanya:
   - Row-level locking (pessimistic)
   - Optimistic concurrency control
3. Lakukan pengujian konkurensi tinggi (minimal 20 koneksi simultan) dan analisis:
   - Throughput (transactions per second)
   - Deadlock yang terjadi (jika ada)
   - Kesalahan konkurensi lainnya
4. Buat laporan yang menjelaskan bagaimana implementasi transaksi mempengaruhi konkurensi tinggi

7 Pertanyaan Diskusi

1. Apa perbedaan antara READ COMMITTED dan REPEATABLE READ? Kapan sebaiknya menggunakan masing-masing?
2. Bagaimana cara mendeteksi dan mengatasi deadlock pada PostgreSQL?
3. Apa trade-off antara optimistic dan pessimistic concurrency control?
4. Bagaimana dampak tingkat isolasi transaksi terhadap performa sistem dengan konkurensi tinggi?
5. Bagaimana strategi yang tepat untuk menangani sistem dengan write-heavy workload?

8 Referensi

- PostgreSQL Documentation: Transactions
- PostgreSQL: Concurrency Control
- PostgreSQL: Explicit Locking
- PostgreSQL: Administrative Functions
- PostgreSQL: Monitoring Database Activity

Bab 3

Implementasi Partitioning pada PostgreSQL

1 Pendahuluan

Partitioning adalah teknik untuk membagi tabel besar menjadi beberapa bagian yang lebih kecil secara fisik, namun tetap terlihat sebagai satu tabel secara logis. Dalam sistem database dengan volume data yang besar, partitioning menjadi strategi penting untuk meningkatkan performa, maintainability, dan skalabilitas.

Ketika volume data tumbuh, performa query dapat menurun signifikan karena database harus memproses lebih banyak data. Dengan partitioning, database dapat membatasi jumlah data yang diakses saat menjalankan query, sehingga meningkatkan efisiensi operasi.

1.1 Tujuan Praktikum

Dilaksanakannya praktikum partitioning, praktikan diharapkan mampu:

1. Memahami konsep partitioning pada database
2. Mengimplementasikan berbagai tipe partitioning pada PostgreSQL
3. Menganalisis dampak partitioning terhadap performa query
4. Mempelajari strategi partitioning yang tepat untuk berbagai jenis data

2 Konsep Dasar Partitioning

2.1 Apa itu Partitioning?

Partitioning adalah teknik untuk membagi tabel besar menjadi beberapa bagian yang lebih kecil secara fisik tetapi tetap terlihat sebagai satu tabel secara logis. Hal ini memungkinkan database untuk mengoptimalkan query dengan hanya mengakses partisi yang relevan dengan query, bukan seluruh tabel.

2.2 Manfaat Partitioning

1. **Peningkatan Performa Query**: Query yang hanya mengakses subset data (misalnya satu bulan dari data time-series) dapat berjalan lebih cepat karena hanya mengakses partisi yang relevan
2. **Efisiensi Maintenance**: Operasi maintenance seperti backup, restore, dan archiving dapat dilakukan pada level partisi
3. **Optimalisasi Penyimpanan**: Data dengan pola akses berbeda dapat disimpan pada media penyimpanan yang berbeda (misalnya data terbaru pada SSD)
4. **Paralelisme**: Query terhadap beberapa partisi dapat dieksekusi secara paralel
5. **Penghapusan Data Efisien**: Menghapus data dalam jumlah besar dapat dilakukan dengan DROP PARTITION daripada DELETE yang lebih lambat

2.3 Kapan Menggunakan Partitioning

Partitioning sangat berguna pada:

1. Tabel dengan volume data sangat besar (puluhan GB atau lebih)
2. Data dengan pola akses yang jelas berdasarkan kolom tertentu
3. Data time-series yang memiliki pola pemasukan dan penghapusan teratur
4. Tabel fakta pada data warehouse
5. Sistem yang memerlukan high-availability untuk data terbaru sementara data historis dapat diakses lebih lambat

2.4 Kapan Tidak Menggunakan Partitioning

Partitioning tidak selalu bermanfaat pada:

1. Tabel dengan ukuran kecil atau menengah yang dapat diproses efisien tanpa partitioning
2. Tabel yang sering di-join dengan tabel lain (trade-off performa join vs. partitioning)
3. Sistem dengan pola query yang sulit diprediksi atau tidak selaras dengan strategi partitioning
4. Ketika overhead administrasi partisi lebih besar dari manfaatnya

3 Jenis-jenis Partitioning di PostgreSQL

PostgreSQL menyediakan beberapa tipe partitioning yang dapat dipilih sesuai dengan karakteristik data dan pola akses:

3.1 Range Partitioning

1. Membagi data berdasarkan rentang nilai pada kolom tertentu
2. Ideal untuk data time-series (tanggal, timestamp), nilai numerik berurutan
3. Sangat efektif untuk query yang mengakses data dalam rentang tertentu
4. Contoh: partisi data penjualan per bulan, per kuartal, atau per tahun

3.2 List Partitioning

1. Membagi data berdasarkan daftar nilai diskrit pada kolom tertentu
2. Cocok untuk kolom dengan nilai terbatas dan terdefinisi
3. Efektif untuk query yang mem-filter berdasarkan nilai spesifik
4. Contoh: partisi berdasarkan region, status pesanan, kategori produk

3.3 Hash Partitioning

1. Membagi data berdasarkan hasil fungsi hash dari nilai kolom
2. Mendistribusikan data secara relatif merata di semua partisi
3. Cocok saat tidak ada pola akses yang jelas atau untuk distribusi beban
4. Kurang optimal untuk query range, tetapi baik untuk paralelisme
5. Contoh: partisi berdasarkan hash dari customer_id untuk distribusi merata

Tipe Partitioning	**Karakteristik dan Penggunaan**
Range	Membagi berdasarkan rentang nilai (tanggal, timestamp, numerik). Ideal untuk data time-series dan query range. Misalnya: data per bulan, per semester, atau per tahun.
List	Membagi berdasarkan daftar nilai diskrit. Cocok untuk kolom dengan nilai terbatas seperti region, status, kategori. Misalnya: data per provinsi, per status pesanan.
Hash	Membagi berdasarkan hasil fungsi hash. Distribusi data merata, cocok untuk paralelisme dan distribusi beban. Misalnya: hash dari customer_id untuk distribusi merata.

Gambar 3.1: Jenis-jenis Partitioning di PostgreSQL dan Penggunaannya

3.4 Sub-partitioning (Multi-level Partitioning)

PostgreSQL juga mendukung partitioning bertingkat (sub-partitioning), di mana partisi dapat dibagi lagi menjadi sub-partisi. Hal ini memungkinkan fleksibilitas lebih tinggi dalam manajemen data:

1. Partisi level pertama bisa menggunakan Range (misalnya, per tahun)
2. Sub-partisi bisa menggunakan List (misalnya, per region atau status)
3. Atau kombinasi lain dari jenis partitioning

4 Best Practices Implementasi Partitioning

1. **Pilih kolom partitioning yang sesuai** dengan pola akses query yang paling umum
2. **Batasi jumlah partisi** menjadi jumlah yang dapat dikelola (tidak terlalu banyak)
3. **Pertimbangkan partitioning default** untuk menangani data yang tidak masuk kategori partisi yang ada
4. **Buat index pada setiap partisi** untuk kolom yang sering digunakan dalam query
5. **Gunakan constraint exclusion** untuk memastikan query optimizer melewatkan partisi yang tidak relevan
6. **Pertimbangkan tools otomatisasi** untuk mengelola partisi yang dibuat secara berkala
7. **Uji performa** untuk memverifikasi bahwa partitioning memberikan manfaat yang diharapkan
8. **Kombinasikan dengan strategi penyimpanan** (tablespace) untuk pengoptimalan lebih lanjut

5 Tahapan Praktikum

5.1 Persiapan

1. Pastikan container Docker PostgreSQL sudah berjalan
2. Gunakan pgAdmin atau PSQL untuk mengakses database
3. Pastikan data sudah diimpor menggunakan script yang disediakan

5.2 Implementasi Range Partitioning

Pada bagian ini, kita akan membuat partisi tabel orders berdasarkan tanggal pesanan (order date):

```
1  -- Buat tabel induk dengan deklarasi partisi
2  CREATE TABLE orders_partitioned (
3    order_id SERIAL,
```

```
    customer_id INTEGER NOT NULL,
    order_date TIMESTAMP NOT NULL,
    status VARCHAR(50) DEFAULT 'pending',
    total_amount DECIMAL(12, 2),
    shipping_address TEXT,
    payment_method VARCHAR(50),
    PRIMARY KEY (order_id, order_date),
    CONSTRAINT valid_status CHECK (status IN ('pending', '
   processing', 'shipped', 'delivered', 'cancelled'))
) PARTITION BY RANGE (order_date);

-- Buat partisi untuk setiap kuartal tahun 2023
CREATE TABLE orders_q1_2023 PARTITION OF
 orders_partitioned
  FOR VALUES FROM ('2023-01-01') TO ('2023-04-01');

CREATE TABLE orders_q2_2023 PARTITION OF
 orders_partitioned
  FOR VALUES FROM ('2023-04-01') TO ('2023-07-01');

CREATE TABLE orders_q3_2023 PARTITION OF
 orders_partitioned
  FOR VALUES FROM ('2023-07-01') TO ('2023-10-01');

CREATE TABLE orders_q4_2023 PARTITION OF
 orders_partitioned
  FOR VALUES FROM ('2023-10-01') TO ('2024-01-01');

-- Import data dari tabel orders asli
INSERT INTO orders_partitioned
SELECT * FROM orders;
```

5.3 Implementasi List Partitioning

Selanjutnya, kita akan mencoba partitioning berdasarkan status pesanan:

```
-- Buat tabel induk dengan deklarasi partisi
CREATE TABLE orders_by_status (
    order_id SERIAL,
    customer_id INTEGER NOT NULL,
    order_date TIMESTAMP NOT NULL,
    status VARCHAR(50) NOT NULL,
    total_amount DECIMAL(12, 2),
```

```
    shipping_address TEXT,
    payment_method VARCHAR(50),
    PRIMARY KEY (order_id, status)
) PARTITION BY LIST (status);

-- Buat partisi untuk setiap status
CREATE TABLE orders_pending PARTITION OF orders_by_status
    FOR VALUES IN ('pending');

CREATE TABLE orders_processing PARTITION OF
 orders_by_status
    FOR VALUES IN ('processing');

CREATE TABLE orders_shipped PARTITION OF orders_by_status
    FOR VALUES IN ('shipped');

CREATE TABLE orders_delivered PARTITION OF
 orders_by_status
    FOR VALUES IN ('delivered');

CREATE TABLE orders_cancelled PARTITION OF
 orders_by_status
    FOR VALUES IN ('cancelled');

-- Import data dari tabel orders asli
INSERT INTO orders_by_status
SELECT * FROM orders;
```

5.4 Implementasi Hash Partitioning

Berikutnya, kita akan mencoba partitioning berdasarkan hash dari customer id:

```
-- Buat tabel induk dengan deklarasi partisi
CREATE TABLE orders_by_customer (
    order_id SERIAL,
    customer_id INTEGER NOT NULL,
    order_date TIMESTAMP NOT NULL,
    status VARCHAR(50) DEFAULT 'pending',
    total_amount DECIMAL(12, 2),
    shipping_address TEXT,
    payment_method VARCHAR(50),
    PRIMARY KEY (order_id, customer_id),
```

```
    CONSTRAINT valid_status CHECK (status IN ('pending', '
    processing', 'shipped', 'delivered', 'cancelled'))
) PARTITION BY HASH (customer_id);

-- Buat 4 partisi
CREATE TABLE orders_by_customer_0 PARTITION OF
    orders_by_customer
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);

CREATE TABLE orders_by_customer_1 PARTITION OF
    orders_by_customer
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);

CREATE TABLE orders_by_customer_2 PARTITION OF
    orders_by_customer
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);

CREATE TABLE orders_by_customer_3 PARTITION OF
    orders_by_customer
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);

-- Import data dari tabel orders asli
INSERT INTO orders_by_customer
SELECT * FROM orders;
```

5.5 Menganalisis Performa Partitioning

Pada bagian ini, kita akan membandingkan performa query antara tabel biasa dan tabel yang menggunakan partitioning:

```
-- Query pada tabel non-partisi
EXPLAIN ANALYZE
SELECT *
FROM orders
WHERE order_date BETWEEN '2023-06-01' AND '2023-06-30';

-- Query pada tabel dengan partisi range
EXPLAIN ANALYZE
SELECT *
FROM orders_partitioned
WHERE order_date BETWEEN '2023-06-01' AND '2023-06-30';
```

5.6 Maintenance Partisi

Selanjutnya, kita akan mempelajari bagaimana melakukan maintenance terhadap partisi:

```
-- Menambahkan partisi baru untuk tahun 2024
CREATE TABLE orders_q1_2024 PARTITION OF
 orders_partitioned
  FOR VALUES FROM ('2024-01-01') TO ('2024-04-01');

-- Menghapus data lama dari partisi tertentu
TRUNCATE TABLE orders_q1_2023;

-- Mengarsipkan partisi lama
ALTER TABLE orders_partitioned DETACH PARTITION
 orders_q1_2023;
-- Sekarang orders_q1_2023 adalah tabel mandiri yang
 dapat diarchive
```

5.7 Implementasi Sub-partitioning (Multi-level Partitioning)

Kita juga akan mengimplementasikan partitioning bertingkat:

```
-- Buat tabel induk dengan deklarasi partisi bertingkat
CREATE TABLE orders_multi_level (
  order_id SERIAL,
  customer_id INTEGER NOT NULL,
  order_date TIMESTAMP NOT NULL,
  status VARCHAR(50) NOT NULL,
  total_amount DECIMAL(12, 2),
  shipping_address TEXT,
  payment_method VARCHAR(50),
  PRIMARY KEY (order_id, order_date, status)
) PARTITION BY RANGE (order_date);

-- Buat partisi kuartal, yang juga akan dipartisi lagi
CREATE TABLE orders_q1_2023_multi PARTITION OF
 orders_multi_level
  FOR VALUES FROM ('2023-01-01') TO ('2023-04-01')
  PARTITION BY LIST (status);

-- Subpartisi berdasarkan status untuk Q1 2023
```

```
CREATE TABLE orders_q1_2023_pending PARTITION OF
 orders_q1_2023_multi
  FOR VALUES IN ('pending');

CREATE TABLE orders_q1_2023_processing PARTITION OF
 orders_q1_2023_multi
  FOR VALUES IN ('processing');

CREATE TABLE orders_q1_2023_shipped PARTITION OF
 orders_q1_2023_multi
  FOR VALUES IN ('shipped');

CREATE TABLE orders_q1_2023_delivered PARTITION OF
 orders_q1_2023_multi
  FOR VALUES IN ('delivered');

CREATE TABLE orders_q1_2023_cancelled PARTITION OF
 orders_q1_2023_multi
  FOR VALUES IN ('cancelled');

-- Import data
INSERT INTO orders_multi_level
SELECT * FROM orders
WHERE order_date BETWEEN '2023-01-01' AND '2023-03-31';
```

6 Tugas Praktikum

1. Implementasikan partitioning pada salah satu tabel besar (orders atau products)
2. Bandingkan performa query antara tabel non-partisi dan tabel dengan partisi untuk:
   - Query yang mengakses satu partisi
   - Query yang mengakses beberapa partisi
   - Query yang mengakses seluruh tabel
3. Implementasikan minimal dua tipe partitioning berbeda (Range, List, atau Hash)
4. Buat script untuk maintenance partisi secara otomatis:

- Menambahkan partisi baru secara otomatis
- Mengarsipkan partisi lama

5. Lakukan benchmark konkuren dan bandingkan throughput pada tabel dengan dan tanpa partisi

7 Pertanyaan Diskusi

1. Kapan sebaiknya menggunakan partitioning dan kapan tidak perlu?
2. Bagaimana strategi partitioning yang tepat untuk:
   - Data time-series dengan volume tinggi
   - Data yang sering diakses berdasarkan wilayah geografis
   - Tabel fakta pada data warehouse
3. Apa perbedaan antara partitioning dan sharding? Kapan sebaiknya menggunakan masing-masing?
4. Apa kelebihan dan kekurangan dari masing-masing tipe partitioning (Range, List, Hash)?
5. Bagaimana partitioning mempengaruhi performa JOIN operations?

8 Referensi

- PostgreSQL Documentation: Table Partitioning
- PostgreSQL: Declarative Partitioning
- PostgreSQL Wiki: Table Partitioning
- PostgreSQL: Partition Pruning
- PostgreSQL: Constraint Exclusion

Bab 4

Backup dan Recovery pada PostgreSQL

1 Pendahuluan

Backup dan recovery merupakan aspek kritis dalam pengelolaan database. Sebagai database administrator, melindungi data dari kehilangan dan kerusakan merupakan tanggung jawab utama. PostgreSQL menyediakan beragam metode backup dan recovery yang dapat disesuaikan dengan kebutuhan sistem, batasan operasional, dan tujuan bisnis.

Strategi backup dan recovery yang tepat menjamin ketahanan (resilience) dan ketersediaan (availability) sistem database. Dalam lingkungan produksi, kemampuan untuk memulihkan data dengan cepat dan akurat saat terjadi kegagalan sistem, kesalahan manusia, atau bencana dapat menjadi perbedaan antara operasi bisnis yang berkelanjutan dan kerugian finansial yang signifikan.

1.1 Tujuan Praktikum

Dilaksanakannya praktikum backup dan recovery, praktikan diharapkan mampu:

1. Memahami konsep backup dan recovery pada database
2. Mengimplementasikan berbagai tipe backup di PostgreSQL
3. Melakukan recovery data dari backup
4. Menerapkan strategi backup yang tepat sesuai kebutuhan

2 Konsep Dasar Backup dan Recovery

2.1 Terminologi dan Konsep Kunci

1. **Backup**: Proses membuat salinan data untuk pemulihan jika terjadi kehilangan data
2. **Recovery**: Proses memulihkan database ke keadaan yang konsisten menggunakan data backup
3. **Recovery Point Objective (RPO)**: Jumlah maksimal data yang dapat hilang, diukur dalam waktu
4. **Recovery Time Objective (RTO)**: Waktu maksimal yang diperbolehkan untuk memulihkan sistem
5. **Full Backup**: Backup seluruh database
6. **Incremental Backup**: Backup hanya perubahan sejak backup terakhir
7. **Differential Backup**: Backup perubahan sejak full backup terakhir
8. **Point-in-Time Recovery (PITR)**: Kemampuan memulihkan ke titik waktu tertentu

2.2 Tipe-tipe Backup di PostgreSQL

PostgreSQL menyediakan beberapa metode backup yang dapat digunakan sesuai kebutuhan:

1. **Logical Backup**:
   - Menggunakan utilitas `pg_dump` dan `pg_dumpall`
   - Menghasilkan file SQL atau format khusus yang berisi perintah untuk merekonstruksi database
   - Fleksibel dan dapat dipulihkan secara selektif
   - Cocok untuk database kecil hingga menengah
2. **Physical Backup**:
   - Salinan langsung dari file data PostgreSQL
   - Lebih cepat untuk database besar

- Membutuhkan akses ke sistem file
- Membutuhkan penghentian database atau konsistensi filesystem

3. **Continuous Archiving**:
    - Menggunakan Write-Ahead Logs (WAL) untuk mencatat semua perubahan
    - Memungkinkan Point-in-Time Recovery (PITR)
    - Kombinasi dari base backup dan WAL archives
    - Cocok untuk database dengan zero/minimal data loss requirement

Aspek	**Logical Backup**	**Physical Backup**	**Continuous Archiving**
Metode	pg_dump, pg_dumpall	File system backup	Base backup + WAL archiving
Kelebihan	Selektif, platform-independent, dapat restore sebagian	Cepat untuk database besar, overhead rendah	Point-in-time recovery, minimal data loss
Kekurangan	Lambat untuk database besar, overhead tinggi	Tidak selektif, platform-dependent	Kompleksitas setup, kebutuhan penyimpanan tinggi
Ideal untuk	Database kecil-menengah, backup selektif	Database besar, full restore cepat	Sistem kritikal, zero data loss requirement

Gambar 4.1: Perbandingan Tipe Backup di PostgreSQL

3 Strategi Backup yang Efektif

3.1 Faktor yang Mempengaruhi Strategi Backup

1. **Volume Data**: Ukuran database mempengaruhi waktu backup dan storage requirements
2. **Window Backup**: Periode waktu yang tersedia untuk melakukan backup
3. **RPO dan RTO**: Persyaratan bisnis untuk toleransi kehilangan data dan waktu pemulihan

4. **Resource Constraints**: Keterbatasan penyimpanan, CPU, dan bandwith jaringan
5. **Frekuensi Perubahan**: Seberapa sering dan banyak data berubah
6. **Retensi Backup**: Berapa lama backup harus disimpan

3.2 Rekomendasi Best Practices

1. **Redundansi**: Simpan beberapa salinan backup di lokasi berbeda
2. **Verifikasi Backup**: Secara teratur menguji integritas backup dengan test restore
3. **Otomatisasi**: Jadwalkan backup secara reguler menggunakan cron jobs atau tools
4. **Monitoring**: Pantau proses backup dan segera tangani kegagalan
5. **Dokumentasi**: Dokumentasikan prosedur backup dan recovery secara detail
6. **Enkripsi**: Lindungi data backup dengan enkripsi jika berisi informasi sensitif
7. **Rotasi Backup**: Implementasikan strategi rotasi untuk mengoptimalkan penyimpanan
8. **Strategi Berlapis**: Kombinasikan beberapa metode backup (misal: full backup mingguan + incremental harian + continuous archiving)

4 Tahapan Praktikum

4.1 Persiapan

1. Pastikan container Docker PostgreSQL sudah berjalan
2. Gunakan pgAdmin atau PSQL untuk mengakses database
3. Pastikan data sudah diimpor menggunakan script yang disediakan

4.2 Logical Backup dengan pg_dump

Mari kita mulai dengan melakukan logical backup menggunakan pg_dump:

Backup Database Lengkap

```
# Format SQL
pg_dump -h localhost -p 5432 -U praktikan -d ecommerce -f
  ecommerce_full.sql

# Format Custom (untuk restore selektif)
pg_dump -h localhost -p 5432 -U praktikan -d ecommerce -
 Fc -f ecommerce_full.dump

# Format Directory (paralel backup)
pg_dump -h localhost -p 5432 -U praktikan -d ecommerce -
 Fd -j 4 -f ecommerce_backup_dir
```

Backup Tabel Tertentu

```
# Backup hanya tabel orders dan order_items
pg_dump -h localhost -p 5432 -U praktikan -d ecommerce -t
  orders -t order_items -f orders_backup.sql
```

Backup Schema Saja

```
# Backup struktur tanpa data
pg_dump -h localhost -p 5432 -U praktikan -d ecommerce --
 schema-only -f ecommerce_schema.sql
```

Backup Data Saja

```
# Backup data tanpa struktur
pg_dump -h localhost -p 5432 -U praktikan -d ecommerce --
 data-only -f ecommerce_data.sql
```

4.3 Restore dari Logical Backup

Setelah melakukan backup, kita juga perlu tahu cara melakukan restore:

Restore Database Lengkap

```
# Restore dari format SQL
psql -h localhost -p 5432 -U praktikan -d
 ecommerce_restore -f ecommerce_full.sql

# Restore dari format Custom
pg_restore -h localhost -p 5432 -U praktikan -d
 ecommerce_restore -v ecommerce_full.dump
```

Restore Tabel Tertentu

```
# Restore hanya tabel orders dari backup custom
pg_restore -h localhost -p 5432 -U praktikan -d
 ecommerce_restore -t orders -v ecommerce_full.dump
```

4.4 Physical Backup

Physical backup melibatkan penyalinan langsung file data PostgreSQL. Dalam container Docker, kita bisa menggunakan volume untuk menyimpan data:

```
# Stop container
docker-compose stop postgres

# Backup volume
docker run --rm -v postgres_praktikum_postgres_data:/
 source -v $(pwd)/backup:/backup ubuntu tar -czf /backup/
 postgres_data_backup.tar.gz -C /source .

# Start container kembali
docker-compose start postgres
```

4.5 Continuous Archiving dan Point-in-Time Recovery (PITR)

Untuk mengimplementasikan Continuous Archiving, kita perlu mengkonfigurasi PostgreSQL untuk menyimpan Write-Ahead Logs (WAL):

Konfigurasi WAL Archiving

Edit file `postgresql.conf`:

```
wal_level = replica
archive_mode = on
archive_command = 'cp %p /var/lib/postgresql/data/archive/%f'
```

Point-in-Time Recovery

```
# Buat base backup
pg_basebackup -h localhost -p 5432 -U praktikan -D /backup/base -Fp -Xs -P

# Restore ke titik waktu tertentu
# (Memerlukan recovery.conf)
```

4.6 Demonstrasi Skenario Disaster

Pada bagian ini, kita akan mensimulasikan skenario kehilangan data dan melakukan recovery:

Simulasi Kehilangan Data

```
-- Buat tabel untuk demonstrasi
CREATE TABLE important_data (
  id SERIAL PRIMARY KEY,
  description TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insert data penting
INSERT INTO important_data (description) VALUES ('Data sangat penting #1');
INSERT INTO important_data (description) VALUES ('Data sangat penting #2');
INSERT INTO important_data (description) VALUES ('Data sangat penting #3');

-- Backup
-- (Jalankan pg_dump seperti di atas)

-- Simulasi kesalahan
DROP TABLE important_data;

```

```
-- Restore
-- (Jalankan pg_restore seperti di atas)
```

Simulasi Corrupt Database

```
# Backup terlebih dahulu
# (Jalankan backup)

# Simulasi korupsi (hati-hati!)
docker-compose exec postgres bash -c "dd if=/dev/urandom
 of=/var/lib/postgresql/data/base/16384/1234 bs=8192
 count=1 conv=notrunc"

# Restart PostgreSQL (kemungkinan akan gagal start)
docker-compose restart postgres

# Restore dari backup
# (Jalankan restore)
```

4.7 Strategi Backup yang Baik

Berdasarkan praktik terbaik, berikut adalah strategi backup yang direkomendasikan:

1. Full backup setiap minggu
2. Differential backup setiap hari
3. Continuous archiving untuk recovery point-in-time
4. Replikasi untuk high availability

4.8 Implementasi High Availability dengan Replikasi

Meskipun bukan bagian dari backup/recovery secara tradisional, replikasi membantu memastikan ketersediaan data:

```
# Primary server (postgresql.conf)
wal_level = replica
max_wal_senders = 10
wal_keep_segments = 64

# Standby server (postgresql.conf)
```

```
hot_standby = on

# Standby server (recovery.conf)
standby_mode = 'on'
primary_conninfo = 'host=primary port=5432 user=
 replicator password=password'
trigger_file = '/tmp/postgresql.trigger'
```

5 Tugas Praktikum

1. Lakukan backup database dengan minimal 3 metode berbeda:
   - Full database backup dengan pg_dump
   - Backup tabel tertentu
   - Backup struktur saja
2. Simulasikan skenario kehilangan data dan lakukan recovery dari backup:
   - Menghapus tabel penting secara tidak sengaja
   - Menghapus beberapa baris data
   - Menghapus database
3. Buat script otomatisasi backup yang berjalan terjadwal:
   - Menggunakan cron job atau scheduled tasks
   - Menyimpan backup dengan timestamp
   - Menghapus backup lama secara otomatis
4. Buat laporan yang menjelaskan strategi backup dan recovery yang tepat untuk database dengan karakteristik:
   - Data transaksi finansial yang kritikal
   - Data logging dengan volume tinggi
   - Data referensi yang jarang berubah

6 Pertanyaan Diskusi

1. Apa kelebihan dan kekurangan masing-masing metode backup (logical vs physical vs continuous archiving)?
2. Bagaimana strategi backup yang tepat untuk database dengan ukuran sangat besar (¿1TB)?
3. Apa trade-off antara keamanan data (durability) dengan performa database?
4. Bagaimana cara menghitung Recovery Point Objective (RPO) dan Recovery Time Objective (RTO) untuk database?
5. Apa saja pertimbangan saat merancang strategi disaster recovery untuk database produksi?

7 Referensi

- PostgreSQL Documentation: Backup and Restore
- PostgreSQL: Continuous Archiving and PITR
- PostgreSQL: High Availability
- PostgreSQL Wiki: Backup in Practice
- PostgreSQL: pg_dump
- PostgreSQL: pg_restore