MODUL PRATIKUM 4 – PERGUDANGAN DATA

Physical Data Warehouse Design

Dimas Dwi Randa, S.Kom., M.Kom

PROGRAM STUDI SAINS DATA

FAKULTAS SAINS INSTITUT TEKNOLOGI SUMATERA

TAHUN AJAR GANJIL 2025/2026

BAB VI

PHYSICAL DESIGN

1. Deskripsi Singkat

Modul pratikum ini kita akan membahas bagaimana merancang sebuah gudang data yang memilik performa yang bagus. Desain fisik (Physical Design) berfokus pada struktur table dengan algoritma-algoritma yang terstruktur.

2. Tujuan Praktikum

Setelah mempelajari ini:

a. Mahasiswa membuat struktur table yang baik.

b. Mahasiswa merancang agar table yang digunakan flexibel.

c. Mahasiswa menganalisis studi kasus.

3. Alat dan Bahan

a. Komputer

b. SQL Server Management

4. Materi Praktikum

4.1 Definisi Fisikal Gudang Data

Desain Fisikal Gudang Data adalah proses pengaturan bagaimana data warehouse diimplemen tasikan secara fisik dalam sistem basis data, mencakup penggunaan materialized views, mekanisme indeks khusus (seperti bitmap dan join index), serta teknik partisi untuk membagi tabel besar menjadi lebih kecil agar query dapat diproses lebih cepat

4.2 Pemodelan Fisikal Gudang Data

Pada bagian ini diperkenalkan tiga teknik utama yang digunakan untuk memperbaiki performa query dalam gudang data, yaitu materialized views, indexing, dan partitioning

4.3 Materialized Views

Materialized View adalah sebuah view yang hasilnya disimpan secara permanen dalam basis ulang operasi join atau agregasi. data, berfungsi sebagai cache yang

Contoh SQL: Membuat Materialized View.

a. Buatkan database dulu

```
CREATE DATABASE SalesDB;
GO
USE SalesDB;
GO
```

```
Commands completed successfully.

Completion time: 2025-10-02T15:01:22.3985500+07:00
```

b. Buat Tabel Sales

```
CREATE TABLE dbo.Sales (
    SalesID INT PRIMARY KEY,
    ProductKey INT NOT NULL,
    CustomerKey INT NOT NULL,
    DateKey DATE NOT NULL,
    Quantity INT NOT NULL,
    SalesAmount DECIMAL(18,2) NOT NULL
);
```

```
Commands completed successfully.

Completion time: 2025-10-02T15:01:55.3128156+07:00
```

c. **Isi Data Dummy**

```
INSERT INTO dbo.Sales (SalesID, ProductKey, CustomerKey, DateKey, Quantity,
SalesAmount)
VALUES
(1, 101, 201, '2022-01-05', 50,  500.00),
(2, 101, 202, '2022-01-15', 60,  600.00),
(3, 101, 203, '2022-01-25', 70,  700.00), -- total 180
(4, 102, 204, '2022-02-10', 40,  400.00),
(5, 102, 205, '2022-02-20', 55,  550.00), -- total 95
(6, 103, 206, '2022-02-25', 160, 1600.00),
(7, 104, 207, '2022-03-01', 20,  200.00),
(8, 104, 208, '2022-03-05', 25,  250.00); -- total 45
```

```
(8 rows affected)

Completion time: 2025-10-02T15:02:27.2744102+07:00
```

d. Buat Indexed View (Materialized View)

```
CREATE VIEW dbo.TopProducts
WITH SCHEMABINDING
AS
SELECT
   ProductKey,
   SUM(Quantity) AS TotalQuantity,
   COUNT_BIG(*) AS TotalRows
FROM dbo.Sales
GROUP BY ProductKey
```

```
HAVING SUM(Quantity) > 150;
GO
```

```
Commands completed successfully.

Completion time: 2025-10-02T15:04:17.1328495+07:00
```

e. Cek Isi Materialized View

```
SELECT * FROM dbo.TopProducts;
```

	ProductKey	TotalQuantity	TotalRows
1	101	180	3
2	103	160	1

f. Jika ingin menghapus databasenya

```
USE master;
GO
DROP DATABASE SalesDB;
GO
```

4.4 Algoritma dengan Informasi Lengkap

Pada algoritma pemeliharaan view dengan informasi lengkap, sistem basis data diasumsikan memiliki akses penuh terhadap tabel dasar maupun materialized view yang sudah tersimpan. Dengan akses penuh ini, sistem dapat melakukan pembaruan view secara tepat karena bisa mengecek langsung sumber data aslinya. Tiga jenis view yang dapat ditangani dengan pendekatan ini adalah view non-rekursif, view dengan outer join, dan view rekursif.

4.4.1 View non-rekursif

View non-rekursif adalah view yang dapat berisi operasi join, union, negasi, maupun agregasi. Contohnya:

Kasus: Penjualan Online

Perusahaan E-Commerce ingin mengetahui:

1. Total transaksi per pelanggan
2. Total transaksi per kota

a. Membuat Database

```
CREATE DATABASE ECommerceDB;
GO
USE ECommerceDB;
GO
```

```
Commands completed successfully.

Completion time: 2025-10-02T15:51:52.9473177+07:00
```

b. Buat Tabel Customer dan Tabel Orders

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    CustomerName VARCHAR(100),
    City VARCHAR(50)
);
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    Amount DECIMAL(10,2),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

```
Commands completed successfully.

Completion time: 2025-10-02T15:52:39.6976542+07:00
```

c. Masukkan Data Dummy Customers dan Orders

INSERT INTO Customers (CustomerID, CustomerName, City) VALUES

(1, 'Andi', 'Jakarta'),

(2, 'Budi', 'Bandung'),

(3, 'Citra', 'Jakarta'),

(4, 'Dewi', 'Surabaya'),

(5, 'Eko', 'Bandung');

INSERT INTO Orders (OrderID, CustomerID, OrderDate, Amount) VALUES

(101, 1, '2025-09-01', 500000),

(102, 1, '2025-09-05', 300000),

(103, 2, '2025-09-07', 450000),

(104, 3, '2025-09-08', 200000),

(105, 4, '2025-09-09', 750000),

(106, 5, '2025-09-10', 150000),

(107, 2, '2025-09-12', 400000),

(108, 3, '2025-09-13', 600000);

```
(5 rows affected)

(8 rows affected)

Completion time: 2025-10-02T15:53:08.7093534+07:00
```

d. View Non-Rekursif

- **View total transaksi per pelanggan**

CREATE VIEW vw_TotalCustomer

```
AS
SELECT
   c.CustomerID,
   c.CustomerName,
   SUM(o.Amount) AS TotalAmount
FROM Customers c
JOIN Orders o ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID, c.CustomerName;
```

```
Commands completed successfully.

Completion time: 2025-10-02T15:53:32.6465005+07:00
```

- **View total transaksi per kota**

```
CREATE VIEW vw_TotalCity
AS
SELECT
   c.City,
   SUM(o.Amount) AS TotalAmount
FROM Customers c
JOIN Orders o ON c.CustomerID = o.CustomerID
GROUP BY c.City;
```

```
Commands completed successfully.

Completion time: 2025-10-02T15:53:56.4998708+07:00
```

e. Query Pemakaian

Lihat total transaksi per pelanggan

```
SELECT * FROM vw_TotalCustomer;
```

	CustomerID	CustomerName	TotalAmount
1	1	Andi	800000.00
2	2	Budi	850000.00
3	3	Citra	800000.00
4	4	Dewi	750000.00
5	5	Eko	150000.00

Lihat total transaksi per kota

SELECT * FROM vw_TotalCity;

	City	TotalAmount
1	Bandung	1000000.00
2	Jakarta	1600000.00
3	Surabaya	750000.00

4.4.2 View Outer Join

Untuk memelihara view, full outer join dapat ditulis ulang sebagai left outer join atau right outer join, tergantung apakah perubahan terjadi di tabel kiri atau kanan. Hasil query kemudian digabungkan dengan view yang ada.

a. Untuk database dan table tetap sama dengan yang view rekursif jadi tidak perlu membuat database dan table baru

b. **View Non-Rekursif dengan OUTER JOIN**

```
CREATE VIEW vw_TotalCustomerOuter
AS
SELECT
    c.CustomerID,
    c.CustomerName,
    c.City,
    ISNULL(SUM(o.Amount), 0) AS TotalAmount
FROM Customers c
```

```
LEFT OUTER JOIN Orders o ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID, c.CustomerName, c.City;
```

```
Commands completed successfully.

Completion time: 2025-10-02T16:02:22.4300203+07:00
```

c. **Query Pemakaian**

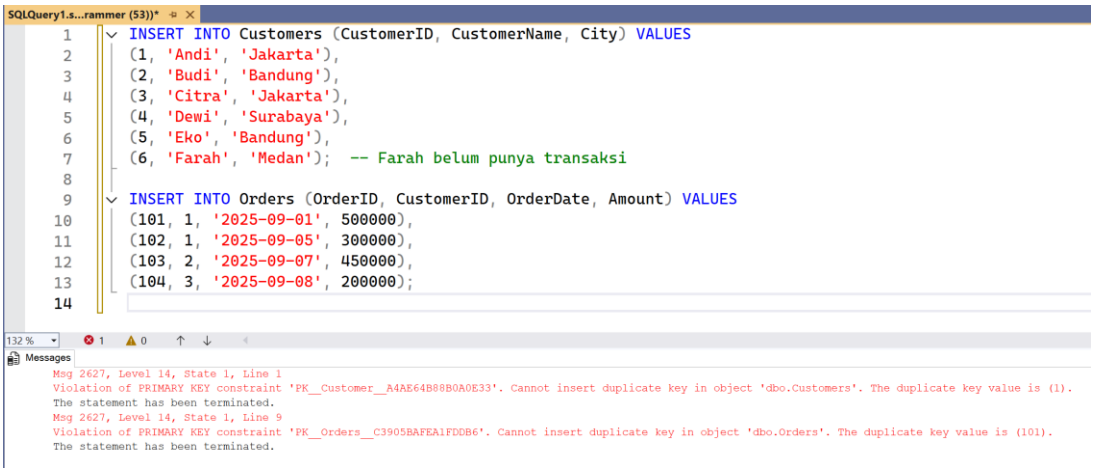
SELECT * FROM vw_TotalCustomerOuter;

	CustomerID	CustomerName	City	TotalAmount
1	1	Andi	Jakarta	800000.00
2	2	Budi	Bandung	850000.00
3	3	Citra	Jakarta	800000.00
4	4	Dewi	Surabaya	750000.00
5	5	Eko	Bandung	150000.00

d. **Masukkan data dummy**

```
INSERT INTO Customers (CustomerID, CustomerName, City) VALUES
(1, 'Andi', 'Jakarta'),
(2, 'Budi', 'Bandung'),
(3, 'Citra', 'Jakarta'),
(4, 'Dewi', 'Surabaya'),
(5, 'Eko', 'Bandung'),
(6, 'Farah', 'Medan');  -- Farah belum punya transaksi
INSERT INTO Orders (OrderID, CustomerID, OrderDate, Amount) VALUES
(101, 1, '2025-09-01', 500000),
(102, 1, '2025-09-05', 300000),
(103, 2, '2025-09-07', 450000),
(104, 3, '2025-09-08', 200000);
```

e. Maka pada saat memasukkan query maka akan terjadi error seperti ini



f. Itu karena ada redundancy data atau ada data duplikat. Nah baiknya kita kosongkan atau hapus datanya.

- Untuk hapus data

  DELETE FROM Orders;

  DELETE FROM Customers;

- Kosongkan table

  TRUNCATE TABLE Orders;

  TRUNCATE TABLE Customers;

Nah harus ingat kondisi ini jika ada datanya yang sama atau double boleh kita delete atau kosongkan tabelnya. Tapi saran saya di hapus aja aja tabelnya dan di masukkan data dummy yang di atas.

g. Jika kita jalankan lagi outer joinnya

SELECT * FROM vw_TotalCustomerOuter;

	CustomerID	CustomerName	City	TotalAmount
1	1	Andi	Jakarta	800000.00
2	2	Budi	Bandung	450000.00
3	3	Citra	Jakarta	200000.00
4	4	Dewi	Surabaya	0.00
5	5	Eko	Bandung	0.00
6	6	Farah	Medan	0.00

h. Nah ada satu cara lagi tanpa kita harus hapus ataupun mengkosongkan data yang lama yaitu kita tambahkan data yang baru

INSERT INTO Customers (CustomerID, CustomerName, City) VALUES

(7, 'Andi', 'Jakarta'),

(8, 'Budi', 'Bandung');

INSERT INTO Orders (OrderID, CustomerID, OrderDate, Amount) VALUES

(201, 7, '2025-09-01', 500000),

(202, 7, '2025-09-05', 300000);

```
(2 rows affected)

(2 rows affected)

Completion time: 2025-10-02T16:12:35.0655364+07:00
```

i. Jika kita jalankan lagi outer joinnya

SELECT * FROM vw_TotalCustomerOuter;

	CustomerID	CustomerName	City	TotalAmount
1	1	Andi	Jakarta	800000.00
2	2	Budi	Bandung	850000.00
3	3	Citra	Jakarta	800000.00
4	4	Dewi	Surabaya	750000.00
5	5	Eko	Bandung	150000.00
6	7	Andi	Jakarta	800000.00
7	8	Budi	Bandung	0.00

Bisa kita lihat bahwa dengan outer join kita tetap bisa memanggil datanya meskipun ada datanya kosong.

4.5 Algoritma dengan Informasi Parsial

Pada banyak kasus, sebuah view tidak dapat selalu dipelihara hanya dengan informasi parsial. Informasi parsial berarti bahwa sistem hanya menggunakan isi materialized view yang sudah ada serta sejumlah aturan integritas (khususnya key constraints), tanpa mengakses tabel dasar secara langsung. Sebuah view disebut self-maintainable apabila ia dapat dipelihara hanya dengan menggunakan view itu sendiri dan key constraints, tanpa perlu mengakses data dasar. Konsep ini sangat penting dalam gudang data karena tujuan utamanya adalah menghindari akses langsung ke data transaksi yang biasanya berukuran sangat besar, khususnya saat memperbarui tabel ringkasan (summary tables).

a. kita masih menggunakan database yang sama dan memiliki data seperti ini

	CustomerID	CustomerName	City	TotalAmount
1	1	Andi	Jakarta	800000.00
2	2	Budi	Bandung	450000.00
3	3	Citra	Jakarta	200000.00
4	4	Dewi	Surabaya	0.00
5	5	Eko	Bandung	0.00
6	6	Farah	Medan	0.00

Andi (Jakarta) → 2 transaksi = 800.000

Budi (Bandung) → 1 transaksi = 450.000

Citra (Jakarta) → 1 transaksi = 200.000

Dewi, Eko, Farah → belum ada transaksi = 0

b. Kalau kita lanjutkan ke **Algoritma dengan Informasi Parsial** Artinya kita tidak tahu jumlah pasti baris dari view sebelum query dijalankan, jadi pakai estimasi berdasarkan data sumber.

c. Estimasi Ukuran View

vw_TotalCustomer → kira-kira = jumlah pelanggan = **6 baris**

vw_TotalCity → kira-kira = jumlah kota unik = **4 baris** (Jakarta, Bandung, Surabaya, Medan)

Dan benar dari hasilmu, vw_TotalCustomer berisi **6 baris** sesuai estimasi

d. Query Tambahan untuk vw_TotalCity

SELECT * FROM vw_TotalCity;

	City	TotalAmount
1	Bandung	450000.00
2	Jakarta	1000000.00

e. Query untuk **total transaksi kota Jakarta**:

SELECT City, TotalAmount

FROM vw_TotalCity

WHERE City = 'Jakarta';

	City	TotalAmount
1	Jakarta	1000000.00

f. Maka dapat kita simpulkan bahwa dengan query tersebut memanggil data yang tersedia saja.

4.6 Pemeliharaan Data Cube

Dalam konteks gudang data, materialized view yang melibatkan fungsi agregasi sering disebut sebagai summary tables. Ringkasnya, summary table adalah hasil perhitungan agregasi atas tabel fakta yang besar, disimpan secara fisik untuk mempercepat query analitis. Tantangan utamanya adalah bagaimana memelihara summary table tersebut agar selalu konsisten dengan data dasar, sambil meminimalkan akses ke data sumber yang biasanya sangat besar.

Contoh Soal: Pemeliharaan Data Cube

Sebuah perusahaan retail menggunakan **SQL Server 2022** untuk membangun *Data Warehouse*. Mereka memiliki **Data Cube Penjualan** yang menyimpan data transaksi bulanan.

1. **Buat Database baru.**

```
CREATE DATABASE RetailDW;
GO
USE RetailDW;
GO
```

2. **Membuat table dimensi**

```
CREATE TABLE dbo.DimWaktu (
    WaktuID INT PRIMARY KEY,
    Tahun INT NOT NULL,
    Bulan INT NOT NULL,
    NamaBulan NVARCHAR(20) NOT NULL
);
CREATE TABLE dbo.DimProduk (
    ProdukID INT PRIMARY KEY,
    NamaProduk NVARCHAR(100) NOT NULL,
    Kategori NVARCHAR(50) NOT NULL
);
CREATE TABLE dbo.DimCabang (
    CabangID INT PRIMARY KEY,
    NamaCabang NVARCHAR(100) NOT NULL,
```

```
    Kota NVARCHAR(50) NOT NULL,
    Provinsi NVARCHAR(50) NOT NULL
);
```

3. **Buat Tabel Fakta**

```
CREATE TABLE dbo.FaktPenjualan (
    PenjualanID INT PRIMARY KEY,
    WaktuID INT NOT NULL,
    ProdukID INT NOT NULL,
    CabangID INT NOT NULL,
    Jumlah INT NOT NULL,
    TotalPenjualan DECIMAL(18,2) NOT NULL,
    FOREIGN KEY (WaktuID) REFERENCES dbo.DimWaktu(WaktuID),
    FOREIGN KEY (ProdukID) REFERENCES dbo.DimProduk(ProdukID),
    FOREIGN KEY (CabangID) REFERENCES dbo.DimCabang(CabangID)
);
```

4. **Isi Data Dimensi table**

```
INSERT INTO dbo.DimWaktu VALUES
(202501, 2025, 1, 'Januari'),
(202502, 2025, 2, 'Februari'),
(202509, 2025, 9, 'September');
INSERT INTO dbo.DimProduk VALUES
(101, 'Laptop Gaming X', 'Elektronik'),
(102, 'Meja Kantor', 'Furniture');
INSERT INTO dbo.DimCabang VALUES
(11, 'Cabang Bandung', 'Bandung', 'Jawa Barat'),
```

(12, 'Cabang Jakarta', 'Jakarta', 'DKI Jakarta');

5. **Isi Data Fakta awal**

```
INSERT INTO dbo.FaktPenjualan VALUES
(1, 202501, 101, 11, 2, 30000000), -- Januari, Bandung, Laptop
(2, 202502, 102, 12, 5, 5000000); -- Februari, Jakarta, Meja
```

6. **Buat Indexed View sebagai Data Cube**

```
IF OBJECT_ID('dbo.vw_CubePenjualan', 'V') IS NOT NULL
   DROP VIEW dbo.vw_CubePenjualan;
GO
```

7. **Create View penjualan**

```
CREATE VIEW dbo.vw_CubePenjualan
WITH SCHEMABINDING
AS
SELECT
   w.Tahun,
   p.Kategori,
   c.Kota,
   SUM(ISNULL(f.TotalPenjualan,0)) AS TotalPenjualan,
   COUNT_BIG(*) AS JumlahTransaksi
FROM dbo.FaktPenjualan f
JOIN dbo.DimWaktu w ON f.WaktuID = w.WaktuID
JOIN dbo.DimProduk p ON f.ProdukID = p.ProdukID
JOIN dbo.DimCabang c ON f.CabangID = c.CabangID
GROUP BY w.Tahun, p.Kategori, c.Kota;
GO
```

8. **Buat clustered index agar view jadi materialized (cube)**

   CREATE UNIQUE CLUSTERED INDEX IX_vw_CubePenjualan

   ON dbo.vw_CubePenjualan (Tahun, Kategori, Kota);

   GO

9. **Cek hasil awal cube**

   SELECT * FROM dbo.vw_CubePenjualan;

   GO

	Tahun	Kategori	Kota	TotalPenjualan	JumlahTransaksi
1	2025	Elektronik	Bandung	30000000.00	1
2	2025	Furniture	Jakarta	5000000.00	1

10. **Tambahkan data transaksi baru (September 2025)**

    INSERT INTO dbo.FaktPenjualan VALUES

    (3, 202509, 101, 11, 5, 75000000), -- 5 laptop di Bandung

    (4, 202509, 101, 11, 3, 45000000); -- 3 laptop di Bandung

11. **Cek lagi hasil cube setelah insert**

    SELECT * FROM dbo.vw_CubePenjualan

    WHERE Tahun = 2025 AND Kategori = 'Elektronik' AND Kota = 'Bandung';

    GO

	Tahun	Kategori	Kota	TotalPenjualan	JumlahTransaksi
1	2025	Elektronik	Bandung	150000000.00	3

12. Jadi, pemeliharaan Data Cube dengan Indexed View adalah **strategi physical design** di SQL Server untuk menjaga performa query agregasi sekaligus memastikan data tetap **up-to-date dan konsisten**.

4.7 Algoritma Klasik dalam Pemeliharaan Data Cube

Pada praktiknya, pemeliharaan summary table yang berbentuk data cube tidak sesederhana menambah atau mengurangi nilai agregasi. Data cube biasanya berisi sejumlah besar agregasi multidimensi yang dihitung dari tabel fakta. Misalnya, sebuah data cube SalesCube dapat menyimpan total penjualan berdasarkan kombinasi dimensi Product, Customer, Store, dan Date. Algoritma Agregasi Inkremental Adalah Algoritma klasik yang digunakan untuk memelihara data cube inkremental mendasarkan diri pada perhitungan delta (Δ) pada setiap level agregasi. Contoh kasus lanjutan yang tadi kita sudah punya Indexed View vw_CubePenjualan sebagai cube. Misalnya, sebelumnya di cube kita punya data:

Tahun	Kategori	Kota	TotalPenjualan	JumlahTransaksi
2025	Elektronik	Bandung	30,000,000	1

1. **Cek data table fakt penjualan dulu**

   SELECT * FROM dbo.FaktPenjualan;

	PenjualanID	WaktuID	ProdukID	CabangID	Jumlah	TotalPenjualan
1	1	202501	101	11	2	30000000.00
2	2	202502	102	12	5	5000000.00
3	3	202509	101	11	5	75000000.00
4	4	202509	101	11	3	45000000.00

2. **Data Baru (Delta Δ)**

   Masuk transaksi September 2025:

   INSERT INTO dbo.FaktPenjualan VALUES

   (5, 202509, 101, 11, 5, 75000000), -- transaksi baru

   (6, 202509, 101, 11, 3, 45000000); -- transaksi baru

   Δ di sini adalah:

   ΔTotalPenjualan = 75,000,000 + 45,000,000 = 120,000,000

   ΔJumlahTransaksi = 2

3. **Query SQL (Tanpa Indexed View – pakai pendekatan manual)**

```
-- Hitung delta transaksi baru (September 2025)
SELECT
    w.Tahun,
    p.Kategori,
    c.Kota,
    SUM(f.TotalPenjualan) AS DeltaTotal,
    COUNT(*) AS DeltaTransaksi
FROM dbo.FaktPenjualan f
JOIN dbo.DimWaktu w ON f.WaktuID = w.WaktuID
JOIN dbo.DimProduk p ON f.ProdukID = p.ProdukID
JOIN dbo.DimCabang c ON f.CabangID = c.CabangID
WHERE f.WaktuID = 202509 -- hanya data baru (September)
GROUP BY w.Tahun, p.Kategori, c.Kota;
```

	Tahun	Kategori	Kota	DeltaTotal	DeltaTransaksi
1	2025	Elektronik	Bandung	240000000.00	4

Hasilnya akan memberi kita nilai Δ **(delta)** yang kemudian ditambahkan ke hasil agregasi lama.

4. **Query SQL (Dengan Indexed View – otomatis)**

Kalau kita pakai Indexed View, SQL Server sudah menghitung **inkremental** secara internal.

```
SELECT * FROM dbo.vw_CubePenjualan
WHERE Tahun = 2025 AND Kategori = 'Elektronik' AND Kota = 'Bandung';
```

	Tahun	Kategori	Kota	TotalPenjualan	JumlahTransaksi
1	2025	Elektronik	Bandung	270000000.00	5

Hasilnya langsung **270,000,000** dan 5 transaksi, tanpa kita hitung manual.

Kesimpulan dari hasil ini

a. Data Cube terpelihara dengan benar

- Semua transaksi yang kamu masukkan (Penjualan laptop di Bandung tahun 2025) sudah terkalkulasi otomatis ke dalam cube.
- Total penjualan yang tercatat = 270 juta dengan 5 transaksi.

b. Algoritma Agregasi Inkremental bekerja

- SQL Server tidak menghitung ulang dari nol, tapi hanya menambahkan delta transaksi baru ke agregasi lama.
- nilah bukti mekanisme *incremental maintenance* dari **Indexed View** (cube materialized).

c. Physical Design terbukti efektif

- Dengan adanya Indexed View, query analitik bisa langsung ambil hasil agregasi tanpa harus GROUP BY ulang setiap kali.
- Ini membuat query lebih cepat dan pemeliharaan cube lebih sederhana.

4.8 Perhitungan dan Materialisasi Data Cube

Materialisasi data cube adalah proses menghitung hasil agregasi pada berbagai level kombinasi dijawab dengan cepat. dimensi dan menyimpannya secara permanen di dalam basis data, sehingga query OLAP dapat dijawab dengan cepat. Berikut contoh soalnya

1. **Buat database baru**

```
CREATE DATABASE RetailCubeDB;
GO
USE RetailCubeDB;
GO
```

2. **Buat table**

```
CREATE TABLE dbo.FaktPenjualan (
    PenjualanID INT PRIMARY KEY,
    Tahun INT NOT NULL,
    Bulan INT NOT NULL,
    Produk NVARCHAR(50) NOT NULL,
    Kota NVARCHAR(50) NOT NULL,
    Jumlah INT NOT NULL,
    TotalPenjualan DECIMAL(18,2) NOT NULL
);
GO
```

3. **Insert data ke table**

```
INSERT INTO dbo.FaktPenjualan (PenjualanID, Tahun, Bulan, Produk, Kota, Jumlah,
TotalPenjualan) VALUES
(1, 2025, 1, 'Laptop', 'Bandung', 2, 30000000),
(2, 2025, 1, 'Laptop', 'Jakarta', 3, 45000000),
(3, 2025, 2, 'Meja',   'Bandung', 5, 5000000),
(4, 2025, 2, 'Laptop', 'Bandung', 1, 15000000),
(5, 2025, 2, 'Meja',   'Jakarta', 2, 2000000);
```

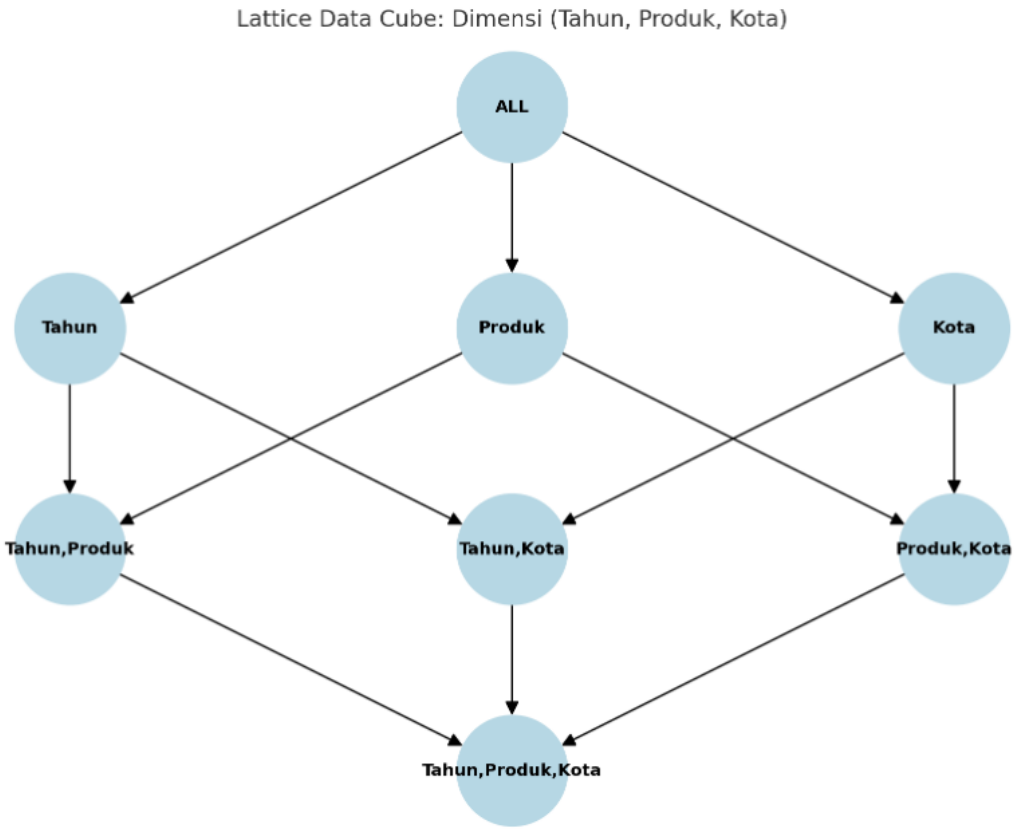
4. **Query perhitungan Data Cube**

```
SELECT
    COALESCE(CAST(Tahun AS VARCHAR(10)), 'ALL') AS Tahun,
    COALESCE(Produk, 'ALL') AS Produk,
    COALESCE(Kota, 'ALL') AS Kota,
    SUM(TotalPenjualan) AS TotalPenjualan,
```

```
    COUNT(*) AS JumlahTransaksi
FROM dbo.FaktPenjualan
GROUP BY CUBE (Tahun, Produk, Kota)
ORDER BY
    CASE WHEN Tahun IS NULL THEN 1 ELSE 0 END, Tahun,
    CASE WHEN Produk IS NULL THEN 1 ELSE 0 END, Produk,
    CASE WHEN Kota IS NULL THEN 1 ELSE 0 END, Kota;
```

	Tahun	Produk	Kota	TotalPenjualan	JumlahTransaksi
1	2025	Laptop	Bandung	45000000.00	2
2	2025	Laptop	Jakarta	45000000.00	1
3	2025	Laptop	ALL	90000000.00	3
4	2025	Meja	Bandung	5000000.00	1
5	2025	Meja	Jakarta	2000000.00	1
6	2025	Meja	ALL	7000000.00	2
7	2025	ALL	Bandung	50000000.00	3
8	2025	ALL	Jakarta	47000000.00	2
9	2025	ALL	ALL	97000000.00	5
10	ALL	Laptop	Bandung	45000000.00	2
11	ALL	Laptop	Jakarta	45000000.00	1
12	ALL	Laptop	ALL	90000000.00	3
13	ALL	Meja	Bandung	5000000.00	1
14	ALL	Meja	Jakarta	2000000.00	1
15	ALL	Meja	ALL	7000000.00	2
16	ALL	ALL	Bandung	50000000.00	3
17	ALL	ALL	Jakarta	47000000.00	2
18	ALL	ALL	ALL	97000000.00	5

5. **Penjelasan dengan Struktur Lattice Data Cube (3 dimensi)**



Itu adalah **lattice data cube** untuk kasus kita dengan **3 dimensi (Tahun, Produk, Kota)**:

a. **Level 0 (ALL)** → total keseluruhan.

b. **Level 1** → agregasi tunggal: Tahun, Produk, Kota.

c. **Level 2** → agregasi pasangan: (Tahun,Produk), (Tahun,Kota), (Produk,Kota).

d. **Level 3** → kombinasi penuh: (Tahun,Produk,Kota).

Dengan begini, kamu ktia tunjukkan bahwa cube selalu memiliki struktur **lattice** seperti di teori, hanya saja jumlah levelnya bergantung pada banyaknya dimensi.

4.8.1 Algoritma PipeSort

Algoritma PipeSort adalah strategi global untuk menghitung data cube. Algoritma ini menggabungkan empat teknik optimisasi yang sudah dibahas sebelumnya, yaitu cache results, amortize scans, pipelined evaluation, dan pemanfaatan urutan atribut.

Contoh Soal: Perhitungan Data Cube dengan Algoritma PipeSort

1. **Buat database baru**

   ```
   CREATE DATABASE PenjualanDB;
   GO
   USE PenjualanDB;
   GO
   ```

2. **Buat table penjualan**

   ```
   CREATE TABLE Penjualan (
      TransID INT PRIMARY KEY,
      Tahun INT,
      Kota VARCHAR(50),
      Produk VARCHAR(50),
      TotalPenjualan DECIMAL(18,2)
   );
   ```

3. **Insert data penjualan**

   ```
   INSERT INTO Penjualan (TransID, Tahun, Kota, Produk, TotalPenjualan) VALUES
   (1, 2025, 'Bandung', 'Laptop', 30000000),
   (2, 2025, 'Bandung', 'Laptop', 15000000),
   (3, 2025, 'Jakarta', 'Laptop', 45000000),
   (4, 2025, 'Bandung', 'Meja',   5000000),
   (5, 2025, 'Jakarta', 'Meja',   2000000);
   ```

4. **Hitung Data Cube dengan PipeSort**

```
SELECT
    Tahun,
    Kota,
    Produk,
    SUM(TotalPenjualan) AS TotalPenjualan
FROM Penjualan
GROUP BY CUBE (Tahun, Kota, Produk)
ORDER BY Tahun, Kota, Produk;
```

Results Messages

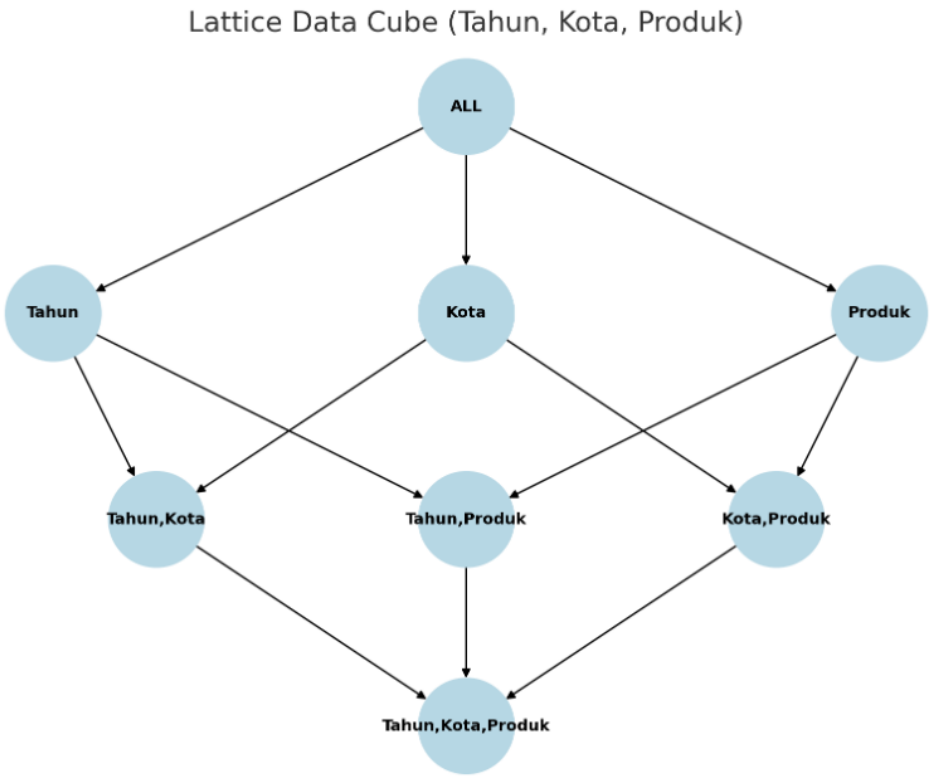
	Tahun	Kota	Produk	TotalPenjualan
1	NULL	NULL	NULL	97000000.00
2	NULL	NULL	Laptop	90000000.00
3	NULL	NULL	Meja	7000000.00
4	NULL	Bandung	NULL	50000000.00
5	NULL	Bandung	Laptop	45000000.00
6	NULL	Bandung	Meja	5000000.00
7	NULL	Jakarta	NULL	47000000.00
8	NULL	Jakarta	Laptop	45000000.00
9	NULL	Jakarta	Meja	2000000.00
10	2025	NULL	NULL	97000000.00
11	2025	NULL	Laptop	90000000.00
12	2025	NULL	Meja	7000000.00
13	2025	Bandung	NULL	50000000.00
14	2025	Bandung	Laptop	45000000.00
15	2025	Bandung	Meja	5000000.00
16	2025	Jakarta	NULL	47000000.00
17	2025	Jakarta	Laptop	45000000.00
18	2025	Jakarta	Meja	2000000.00

5. **Kesimpulan dari contoh**

NULL = menandakan agregasi pada level yang lebih tinggi (*semua nilai*).

Misalnya (2025, Bandung, NULL) artinya total penjualan **semua produk di Bandung pada tahun 2025.**

6. **Diagram lattice Data Cube** untuk dimensi: **Tahun, Kota, Produk**.



ALL (level 0) → total keseluruhan

Level 1 → agregasi berdasarkan 1 dimensi (Tahun, Kota, Produk)

Level 2 → agregasi berdasarkan 2 dimensi (Tahun+Kota, Tahun+Produk, Kota+Produk)

Level 3 → kombinasi penuh (Tahun+Kota+Produk)

4.8.2 Algoritma Bottom-Up Computation (BUC)

Algoritma Bottom-Up Computation (BUC) merupakan metode klasik yang dikembangkan untuk menghitung data cube secara efisien, terutama pada dataset dengan banyak dimensi namun bersifat sparse atau jarang terisi. Tidak seperti PipeSort yang mengandalkan urutan sort

atribut untuk memanfaatkan evaluasi pipelined, BUC bekerja secara rekursif dengan pendekatan divide-and-conquer. Prinsipnya adalah memulai perhitungan dari data detail, lalu secara bertahap menghasilkan agregasi yang lebih umum, sehingga cube dibangun dari bawah ke atas.

Implementasi algoritma BUC di SQL Server 2022

Walaupun BUC aslinya adalah algoritma, di **SQL Server** kita bisa mensimulasikan perhitungannya dengan query **bertingkat (bottom-up)**:

1. **Level 3 (semua dimensi penuh)**

   SELECT Tahun, Kota, Produk, SUM(TotalPenjualan) AS TotalPenjualan

   FROM Penjualan

   GROUP BY Tahun, Kota, Produk;

2. **Level 2 (hilangkan 1 dimensi)**

   - Group berdasarkan tahun dan kota

   SELECT Tahun, Kota, SUM(TotalPenjualan) AS TotalPenjualan

   FROM Penjualan

   GROUP BY Tahun, Kota;

   - Group berdasarkan tahun dan produk

     SELECT Tahun, Produk, SUM(TotalPenjualan) AS TotalPenjualan

     FROM Penjualan

     GROUP BY Tahun, Produk;

   - Group berdasarkan kota dan produk

     SELECT Kota, Produk, SUM(TotalPenjualan) AS TotalPenjualan

     FROM Penjualan

     GROUP BY Kota, Produk;

3. **Level 1 (hilangkan 2 dimensi)**

   - Group berdasarkan tahun

     SELECT Tahun, SUM(TotalPenjualan) AS TotalPenjualan

     FROM Penjualan

     GROUP BY Tahun;

   - Group berdasarkan Kota

     SELECT Kota, SUM(TotalPenjualan) AS TotalPenjualan

     FROM Penjualan

     GROUP BY Kota;

   - Group berdasarkan Produk

     SELECT Produk, SUM(TotalPenjualan) AS TotalPenjualan

     FROM Penjualan

     GROUP BY Produk;

4. **Level 0 (semua ALL → total keseluruhan)**

   SELECT SUM(TotalPenjualan) AS GrandTotal

   FROM Penjualan;

Kesimpulan

a. **BUC** menghitung cube **secara bertahap (bottom-up)** mulai dari kombinasi dimensi terendah, lalu mengagregasi ke atas.

b. Pendekatan ini lebih efisien untuk dataset besar yang **sparse** (banyak nilai kosong), karena tidak perlu menghitung seluruh kombinasi.

c. Di **SQL Server 2022**, BUC dapat disimulasikan dengan **serangkaian query GROUP BY** seperti di atas.

4.8.3 Estimasi Ukuran Data Cube

Sebelumnya telah disebutkan bahwa algoritma seperti PipeSort, dan sebagian besar algoritma lain yang digunakan untuk menghitung summary tables, membutuhkan pengetahuan mengenai ukuran masing masing agregasi. Dengan kata lain, kita harus mengetahui berapa banyak tuple atau baris yang akan dihasilkan oleh setiap kombinasi dimensi. Masalahnya, dalam praktik ukuran ini biasanya tidak diketahui sebelumnya. Oleh karena itu, diperlukan metode untuk memprediksi ukuran agregasi dalam cube dengan cukup akurat. Terdapat tiga metode klasik yang biasa digunakan, meskipun teknik statistik lainnya juga dapat dipakai. Tiga metode tersebut adalah: (1) metode analitis, (2) metode berbasis sampling, dan (3) metode berbasis probabilistic counting.

contoh SQL Server 2022 query dengan 3 pendekatan tadi: **analitis, sampling, probabilistic counting**.

1. **Buat Database CubeEstimationDB**

```
CREATE DATABASE CubeEstimationDB;
GO
USE CubeEstimationDB;
GO
```

2. **Buat tabel fakta Penjualan**

```
CREATE TABLE Penjualan (
    TransID INT PRIMARY KEY,
    Tahun INT,
    Kota VARCHAR(50),
    Produk VARCHAR(50),
    TotalPenjualan DECIMAL(18,2)
);
```

3. **Masukkan data contoh (sederhana)**

   INSERT INTO Penjualan (TransID, Tahun, Kota, Produk, TotalPenjualan) VALUES

   (1, 2024, 'Jakarta', 'Laptop', 3000000),

   (2, 2024, 'Jakarta', 'Meja',   2000000),

   (3, 2024, 'Bandung', 'Laptop', 1500000),

   (4, 2025, 'Jakarta', 'Printer',5000000),

   (5, 2025, 'Surabaya', 'Laptop', 4000000),

   (6, 2025, 'Bandung', 'Kursi', 1000000),

   (7, 2025, 'Bandung', 'Smartphone', 7000000),

   (8, 2025, 'Surabaya', 'Meja',   2500000),

   (9, 2025, 'Jakarta', 'Laptop', 3500000),

   (10,2025, 'Bandung', 'Laptop', 2000000);

4. **Metode Analitis (Distinct Count)**

   a. Jumlah kombinasi Tahun, Kota, Produk

      SELECT COUNT(DISTINCT CONCAT(Tahun, '-', Kota, '-', Produk)) AS Kombinasi_TahunKotaProduk

      FROM Penjualan;

   b. Distinct tiap dimensi

      SELECT COUNT(DISTINCT Tahun) AS Distinct_Tahun,

          COUNT(DISTINCT Kota) AS Distinct_Kota,

          COUNT(DISTINCT Produk) AS Distinct_Produk

      FROM Penjualan;

5. **Metode Sampling**

   Gunakan TABLESAMPLE untuk ambil subset data lalu estimasi.

   a. Ambil 30% sampel data

```
SELECT *
FROM Penjualan TABLESAMPLE (30 PERCENT);
```

b. Ambil 30% baris acak dengan NEWID()

```
SELECT COUNT(DISTINCT CONCAT(CAST(Tahun AS VARCHAR(10)), '-',
Kota, '-', Produk)) AS Kombinasi_Sampel
FROM (
  SELECT TOP 30 PERCENT *
  FROM Penjualan
  ORDER BY NEWID()
) AS Sampel;
```

	Kombinasi_Sampel
1	3

6. **Metode Probabilistic Counting (Simulasi dengan HASHBYTES)**

SQL Server tidak punya HyperLogLog bawaan, tapi bisa disimulasikan dengan hash untuk distinct estimation.

a. Hash setiap produk untuk simulasi probabilistic counting

```
SELECT Produk,
    HASHBYTES('SHA1', Produk) AS ProdukHash
FROM Penjualan;
```

b. Hitung distinct Produk (sebenarnya pakai COUNT DISTINCT, tapi probabilistic counting biasanya hanya estimasi dari hash)

```
SELECT COUNT(DISTINCT Produk) AS DistinctProduk
FROM Penjualan;
```

Kesimpulan

- ✓ **Analitis:** Query COUNT (DISTINCT ...) → hasil presisi.

- ✓ **Sampling:** TABLESAMPLE → estimasi cepat, hasil berbeda tiap kali.
- ✓ **Probabilistic counting:** Simulasi hash → ide bahwa kita tidak butuh full scan untuk estimasi distinct.

4.8.4 Perhitungan Parsial Data Cube

Secara umum, terdapat tiga pendekatan dalam mengimplementasikan sebuah gudang data. Pertama, mematerialisasi seluruh data cube, yaitu menyimpan semua kemungkinan kombinasi agregasi dimensi. Kedua, mematerialisasi sebagian view dari cube, yaitu hanya menyimpan subset view tertentu yang dianggap penting. Ketiga, tidak mematerialisasi sama sekali, artinya semua query harus dihitung langsung dari tabel fakta.

1. **Algoritma Benefit Materialisasi View**

   Contoh ini masih menggunakan database yang sebelumnya namun kita menambahkan table data yang baru.

   a. **Buat table dummy EstimasiView**

      Tabel untuk menyimpan estimasi ukuran (jumlah baris) tiap view

      ```
      CREATE TABLE EstimasiView (
          ViewName VARCHAR(50) PRIMARY KEY,
          Ukuran INT
      );
      ```

   b. **Insert ke table EstimasiView**

      ```
      INSERT INTO EstimasiView (ViewName, Ukuran) VALUES
      ('Tahun', 10),
      ('Kota', 50),
      ('Produk', 100),
      ('TahunKota', 200),
      ```

('TahunProduk', 500),

('KotaProduk', 800),

('TahunKotaProduk', 5000);

c. Hitung biaya tanpa materialisasi

Turunan dari (Tahun, Kota) adalah: Tahun, Kota, TahunKotaProduk

SELECT SUM(Ukuran) AS Biaya_TanpaMaterialisasi

FROM EstimasiView

WHERE ViewName IN ('Tahun', 'Kota', 'TahunKotaProduk');

d. Hitung biaya dengan materialisasi TahunKota

Kita asumsikan:

Biaya materialisasi = ukuran view (TahunKota) → 200

Turunan dihitung 50% lebih murah.

SELECT

200 AS Biaya_Materialisasi_TahunKota,

(SELECT Ukuran * 0.5 FROM EstimasiView WHERE ViewName = 'Tahun') AS Biaya_Tahun,

(SELECT Ukuran * 0.5 FROM EstimasiView WHERE ViewName = 'Kota') AS Biaya_Kota,

(SELECT Ukuran * 0.5 FROM EstimasiView WHERE ViewName = 'TahunKotaProduk') AS Biaya_TahunKotaProduk;

	Biaya_Materialisasi_TahunKota	Biaya_Tahun	Biaya_Kota	Biaya_TahunKotaProduk
1	200	5.0	25.0	2500.0

e. Hitung Benefit

Benefit = biaya tanpa materialisasi – biaya dengan materialisasi

SELECT

(5060 – 2725) AS Benefit_TahunKota;

	Benefit_TahunKota
1	2335

Jadi query di atas mensimulasikan 35able35g algoritma Benefit Materialisasi View di SQL Server 2022.

2. Algoritma Seleksi View Greedy

a. Buat 35able estimasi ukuran view (versi Greedy)

```
CREATE TABLE EstimasiViewGreedy (
    ViewName VARCHAR(50) PRIMARY KEY,
    Ukuran INT
);
```

b. Insert data estimasi ukuran

```
INSERT INTO EstimasiViewGreedy (ViewName, Ukuran) VALUES
('Tahun', 10),
('Kota', 50),
('Produk', 100),
('TahunKota', 200),
('TahunProduk', 500),
('KotaProduk', 800),
('TahunKotaProduk', 5000);
```

c. Contoh hitung benefit untuk View TahunKota

```
SELECT
    (10 + 50 + 5000) AS Biaya_Tanpa,
    (200 + (10*0.5) + (50*0.5) + (5000*0.5)) AS Biaya_Dengan,
    (10 + 50 + 5000) – (200 + (10*0.5) + (50*0.5) + (5000*0.5)) AS Benefit
```

FROM EstimasiViewGreedy

WHERE ViewName = ‘TahunKota’;

	Biaya_Tanpa	Biaya_Dengan	Benefit
1	5060	2730.0	2330.0

d. Contoh hitung benefit untuk View TahunProduk

SELECT

(10 + 100 + 5000) AS Biaya_Tanpa,

(500 + (10*0.5) + (100*0.5) + (5000*0.5)) AS Biaya_Dengan,

(10 + 100 + 5000) – (500 + (10*0.5) + (100*0.5) + (5000*0.5)) AS Benefit

FROM EstimasiViewGreedy

WHERE ViewName = ‘TahunProduk’;

e. Contoh hitung benefit untuk View KotaProduk

SELECT

(50 + 100 + 5000) AS Biaya_Tanpa,

(800 + (50*0.5) + (100*0.5) + (5000*0.5)) AS Biaya_Dengan,

(50 + 100 + 5000) – (800 + (50*0.5) + (100*0.5) + (5000*0.5)) AS Benefit

FROM EstimasiViewGreedy

WHERE ViewName = ‘KotaProduk’;

Dari hasil query di atas, nanti tinggal kita bandingkan nilai **Benefit** untuk tiap kandidat view. Dengan anggaran hanya **2 view**, algoritma seleksi view greedy memilih: (TahunKota) dan (TahunProduk) karena keduanya memberi benefit terbesar.

4.5 Indeks untuk Gudang Data

Salah satu perhatian utama dalam sistem manajemen basis data (DBMS) adalah menyediakan akses data yang cepat. Setiap kali sebuah query diajukan, DBMS relasional berusaha memilih jalur akses (access path) terbaik untuk mendapatkan data. Teknik yang paling populer untuk mempercepat akses data adalah dengan menggunakan indeks. Indeks menyediakan cara cepat untuk menemukan data yang diinginkan. Hampir semua query yang menanyakan data dengan kondisi tertentu akan dijawab dengan bantuan indeks. Sebagai contoh, perhatikan query SQL berikut:

SELECT * FROM Employee WHERE EmployeeKey = 1234

Tanpa indeks pada atribut EmployeeKey, sistem harus melakukan pemindaian penuh (full table scan) terhadap seluruh tabel Employee (kecuali tabel sudah diurutkan). Namun, dengan adanya indeks pada atribut tersebut, DBMS cukup melakukan satu kali akses blok disk untuk menemukan tuple dengan EmployeeKey = 1234, karena atribut ini merupakan kunci dari relasi.

4.5.1 Bitmap Indexes

Untuk membangun bitmap index, dibuat vektor bit dengan panjang sesuai jumlah baris tabel (enam baris dalam contoh ini) untuk setiap nilai atribut

Contoh Latihan: Bitmap Index

Sebuah tabel **Penjualan** memiliki data sebagai berikut:

ID	Produk	Kota
1	Laptop	Jakarta
2	Handphone	Bandung
3	Laptop	Surabaya
4	Tablet	Jakarta
5	Laptop	Bandung

6	Handphone	Surabaya

1. **Buat Database LatihanBitmap**

   CREATE DATABASE LatihanBitmap;

   GO

   USE LatihanBitmap;

   GO

2. **Buat tabel Penjualan**

   CREATE TABLE Penjualan (

   ID INT PRIMARY KEY,

   Produk VARCHAR(50),

   Kota VARCHAR(50)

   );

3. **Insert data contoh**

   INSERT INTO Penjualan (ID, Produk, Kota) VALUES

   (1, ‘Laptop’, ‘Jakarta’),

   (2, ‘Handphone’, ‘Bandung’),

   (3, ‘Laptop’, ‘Surabaya’),

   (4, ‘Tablet’, ‘Jakarta’),

   (5, ‘Laptop’, ‘Bandung’),

   (6, ‘Handphone’, ‘Surabaya’);

4. Simulasi **Bitmap Index** dengan View

   CREATE VIEW BitmapIndex_Produk AS

   SELECT

   ID,

```
    Produk,
    Kota,
    CASE WHEN Produk = ‘Laptop’ THEN 1 ELSE 0 END AS Bitmap_Laptop,
    CASE WHEN Produk = ‘Handphone’ THEN 1 ELSE 0 END AS Bitmap_Handphone,
    CASE WHEN Produk = ‘Tablet’ THEN 1 ELSE 0 END AS Bitmap_Tablet
FROM Penjualan;
```

5. **Lihat hasilnya:**

```
SELECT * FROM BitmapIndex_Produk;
```

	ID	Produk	Kota	Bitmap_Laptop	Bitmap_Handphone	Bitmap_Tablet
1	1	Laptop	Jakarta	1	0	0
2	2	Handphone	Bandung	0	1	0
3	3	Laptop	Surabaya	1	0	0
4	4	Tablet	Jakarta	0	0	1
5	5	Laptop	Bandung	1	0	0
6	6	Handphone	Surabaya	0	1	0

6. **Simulasi Bitmap Index untuk Kota**

```
CREATE VIEW BitmapIndex_Kota AS
SELECT
    ID,
    Produk,
    Kota,
    CASE WHEN Kota = 'Jakarta' THEN 1 ELSE 0 END AS Bitmap_Jakarta,
    CASE WHEN Kota = 'Bandung' THEN 1 ELSE 0 END AS Bitmap_Bandung,
    CASE WHEN Kota = 'Surabaya' THEN 1 ELSE 0 END AS Bitmap_Surabaya
FROM Penjualan;
```

7. **Cek hasilnya**

```
SELECT * FROM BitmapIndex_Kota;
```

	ID	Produk	Kota	Bitmap_Jakarta	Bitmap_Bandung	Bitmap_Surabaya
1	1	Laptop	Jakarta	1	0	0
2	2	Handphone	Bandung	0	1	0
3	3	Laptop	Surabaya	0	0	1
4	4	Tablet	Jakarta	1	0	0
5	5	Laptop	Bandung	0	1	0
6	6	Handphone	Surabaya	0	0	1

8. **Contoh Query dengan Bitmap**

- **Cari semua baris dengan Produk = Laptop**

  SELECT * FROM BitmapIndex_Produk WHERE Bitmap_Laptop = 1;

- **Cari semua baris dengan Kota = Bandung OR Surabaya**

  SELECT * FROM BitmapIndex_Kota

  WHERE Bitmap_Bandung = 1 OR Bitmap_Surabaya = 1;

- **Cari semua baris dengan Produk = Laptop AND Kota = Jakarta**

  SELECT p.*

  FROM BitmapIndex_Produk p

  JOIN BitmapIndex_Kota k ON p.ID = k.ID

  WHERE p.Bitmap_Laptop = 1 AND k.Bitmap_Jakarta = 1;

4.5.2 Kompresi Bitmap

Seperti yang telah kita lihat, bitmap index umumnya bersifat sparse, artinya vektor bit memiliki sedikit nilai ‘1’ di antara banyak nilai ‘0’. Karakteristik ini membuat bitmap index sangat cocok untuk dikompresi.

1. Untuk Database, table dan isian data masih sama dengan contoh yang diatas jadi tidak perlu dibuat
2. Kita buat **bitmap untuk Produk**, simpan sebagai **VARBINARY**. Ide: setiap bit mewakili satu baris tabel.

   -- Membuat bitmap kompresi untuk Produk

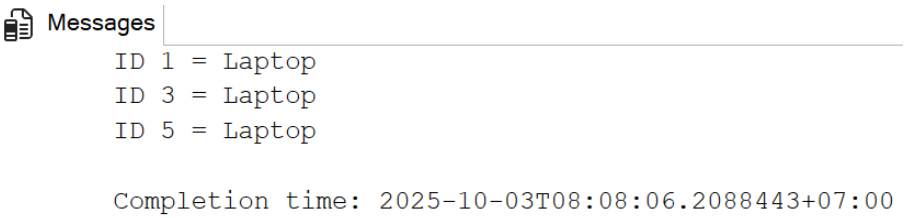
```
DECLARE @BitmapLaptop VARBINARY(8) = 0x00;
-- Loop untuk membangun bitmap
DECLARE @i INT = 1;
WHILE @i <= 6
BEGIN
   DECLARE @b BIT;
   SELECT @b = CASE WHEN Produk = 'Laptop' THEN 1 ELSE 0 END
   FROM Penjualan WHERE ID = @i;
   -- Geser bitmap dan tambahkan bit baru
   SET @BitmapLaptop = CAST(CAST(@BitmapLaptop AS BIGINT) | (@b *
POWER(2, @i-1)) AS VARBINARY(8));
   SET @i = @i + 1;
END
SELECT @BitmapLaptop AS BitmapLaptopKompresi;
```

	BitmapLaptopKompresi
1	0x0000000000000015

3. Query Menggunakan Kompresi Bitmap, Misalnya ingin cek baris mana yang **Produk = Laptop**:

```
-- Dekompresi bitmap dan tampilkan ID yang sesuai
DECLARE @Bitmap VARBINARY(8) = 0x15;  -- hasil bitmap Laptop
DECLARE @i INT = 1;
WHILE @i <= 6
BEGIN
   IF ((CAST(@Bitmap AS BIGINT) & POWER(2, @i-1)) <> 0)
      PRINT 'ID ' + CAST(@i AS VARCHAR(10)) + ' = Laptop';
```

```
    SET @i = @i + 1;
END
```



4.5.3 Join Indexes

Sudah menjadi fakta umum bahwa operasi JOIN merupakan salah satu operasi paling mahal dalam basis data. Join indexes adalah teknik yang sangat efisien untuk mempercepat pemrosesan join, khususnya pada query analitik (decision-support queries). Kekuatan utama join index berasal dari sifat star schema. Dalam star schema, tabel fakta terhubung dengan tabel dimensi melalui foreign key, dan join biasanya dilakukan berdasarkan kunci tersebut. Dengan prekomputasi join ke dalam struktur indeks, sistem dapat menghindari biaya komputasi join berulang kali.

1. **Buatkan database baru LatihanJoinIndex**

```
CREATE DATABASE LatihanJoinIndex;
GO
USE LatihanJoinIndex;
GO
```

2. **Tabel Produk**

```
CREATE TABLE Produk (
    ProdukID INT PRIMARY KEY,
    NamaProduk VARCHAR(50),
    Harga INT
);
```

3. **Tabel Penjualan**

```
CREATE TABLE Penjualan (
    PenjualanID INT PRIMARY KEY,
    ProdukID INT,
    Jumlah INT,
    Kota VARCHAR(50),
    FOREIGN KEY (ProdukID) REFERENCES Produk(ProdukID)
);
```

4. **Insert data Produk**

```
INSERT INTO Produk VALUES
(1, 'Laptop', 10000),
(2, 'Handphone', 5000),
(3, 'Tablet', 7000);
```

5. **Insert data Penjualan**

```
INSERT INTO Penjualan VALUES
(1, 1, 5, 'Jakarta'),
(2, 2, 10, 'Bandung'),
(3, 1, 3, 'Surabaya'),
(4, 3, 7, 'Jakarta'),
(5, 1, 2, 'Bandung'),
(6, 2, 8, 'Surabaya');
```

6. **Buat Indexed View sebagai Join Index**

```
-- Agar bisa membuat Indexed View, harus SET NOCOUNT ON
SET ANSI_NULLS ON;
SET ANSI_PADDING ON;
```

```
SET ANSI_WARNINGS ON;
SET CONCAT_NULL_YIELDS_NULL ON;
SET QUOTED_IDENTIFIER ON;
SET NUMERIC_ROUNDABORT OFF;
GO
-- Buat Indexed View untuk Join Produk + Penjualan
CREATE VIEW JoinIndex_PenjualanProduk
WITH SCHEMABINDING
AS
SELECT
   p.ProdukID,
   p.NamaProduk,
   p.Harga,
   s.PenjualanID,
   s.Jumlah,
   s.Kota
FROM dbo.Produk p
INNER JOIN dbo.Penjualan s
   ON p.ProdukID = s.ProdukID;
GO
-- Tambahkan index supaya materialized view ini cepat
CREATE UNIQUE CLUSTERED INDEX IDX_JoinIndex_PenjualanProduk
ON JoinIndex_PenjualanProduk(PenjualanID);
```

7. Contoh Query Menggunakan Join Index

a. Query total penjualan per Produk

```
SELECT NamaProduk, SUM(Jumlah) AS TotalTerjual
FROM JoinIndex_PenjualanProduk
GROUP BY NamaProduk;
```

	NamaProduk	TotalTerjual
1	Handphone	18
2	Laptop	10
3	Tablet	7

b. Query penjualan Laptop di Jakarta

```
SELECT *
FROM JoinIndex_PenjualanProduk
WHERE NamaProduk = 'Laptop' AND Kota = 'Jakarta';
```

	ProdukID	NamaProduk	Harga	PenjualanID	Jumlah	Kota
1	1	Laptop	10000	1	5	Jakarta

4.6 Evaluasi Query Bintang (StarQueries)

Query yang dijalan kan pada skema Bintang disebut star queries, karena secara eksplisit memanfaatkan struktur star schema: tabel fakta di-join dengan table dimensi.

1. **Buat database LatihanStarSchema**

```
CREATE DATABASE LatihanStarSchema;
GO
USE LatihanStarSchema;
GO
```

2. **Tabel Dimensi Produk**

```
CREATE TABLE DimProduk (
    ProdukID INT PRIMARY KEY,
    NamaProduk VARCHAR(50),
    Kategori VARCHAR(50)
```

```
);
```

3. **Tabel Dimensi Waktu**

```
CREATE TABLE DimWaktu (
    WaktuID INT PRIMARY KEY,
    Tanggal DATE,
    Bulan INT,
    Tahun INT
);
```

4. **Tabel Dimensi Kota**

```
CREATE TABLE DimKota (
    KotaID INT PRIMARY KEY,
    NamaKota VARCHAR(50),
    Provinsi VARCHAR(50)
);
```

5. **Tabel Fakta Penjualan**

```
CREATE TABLE FaktaPenjualan (
    PenjualanID INT PRIMARY KEY,
    ProdukID INT,
    WaktuID INT,
    KotaID INT,
    Jumlah INT,
    Harga INT,
    FOREIGN KEY (ProdukID) REFERENCES DimProduk(ProdukID),
    FOREIGN KEY (WaktuID) REFERENCES DimWaktu(WaktuID),
    FOREIGN KEY (KotaID) REFERENCES DimKota(KotaID)
```

);

6. **Masukkan Data Contoh**

```
-- Dimensi Produk
INSERT INTO DimProduk VALUES
(1, 'Laptop', 'Elektronik'),
(2, 'Handphone', 'Elektronik'),
(3, 'Tablet', 'Elektronik');
-- Dimensi Waktu
INSERT INTO DimWaktu VALUES
(1, '2025-10-01', 10, 2025),
(2, '2025-10-02', 10, 2025),
(3, '2025-10-03', 10, 2025);
-- Dimensi Kota
INSERT INTO DimKota VALUES
(1, 'Jakarta', 'DKI Jakarta'),
(2, 'Bandung', 'Jawa Barat'),
(3, 'Surabaya', 'Jawa Timur');
-- Fakta Penjualan
INSERT INTO FaktaPenjualan VALUES
(1, 1, 1, 1, 5, 10000),
(2, 2, 1, 2, 10, 5000),
(3, 1, 2, 3, 3, 10000),
(4, 3, 3, 1, 7, 7000),
(5, 1, 3, 2, 2, 10000),
(6, 2, 2, 3, 8, 5000);
```

7. Contoh **Star Query**

a. Total Penjualan per Produk

```
SELECT p.NamaProduk, SUM(f.Jumlah * f.Harga) AS TotalPendapatan
FROM FaktaPenjualan f
JOIN DimProduk p ON f.ProdukID = p.ProdukID
GROUP BY p.NamaProduk;
```

b. Total Penjualan per Kota per Bulan

```
SELECT k.NamaKota, w.Bulan, w.Tahun, SUM(f.Jumlah * f.Harga) AS TotalPendapatan
FROM FaktaPenjualan f
JOIN DimKota k ON f.KotaID = k.KotaID
JOIN DimWaktu w ON f.WaktuID = w.WaktuID
GROUP BY k.NamaKota, w.Bulan, w.Tahun;
```

c. Star Query dengan Filter Penjualan Laptop di Jakarta pada Oktober 2025

```
SELECT f.PenjualanID, p.NamaProduk, k.NamaKota, w.Tanggal, f.Jumlah, f.Harga
FROM FaktaPenjualan f
JOIN DimProduk p ON f.ProdukID = p.ProdukID
JOIN DimKota k ON f.KotaID = k.KotaID
JOIN DimWaktu w ON f.WaktuID = w.WaktuID
WHERE p.NamaProduk = 'Laptop'
  AND k.NamaKota = 'Jakarta'
  AND w.Bulan = 10 AND w.Tahun = 2025;
```

	PenjualanID	NamaProduk	NamaKota	Tanggal	Jumlah	Harga
1	1	Laptop	Jakarta	2025-10-01	5	10000

Penjelasan

- ✓ Fakta = FaktaPenjualan
- ✓ Dimensi = DimProduk, DimWaktu, DimKota
- ✓ Query yang **join fakta dengan dimensi** disebut **Star Query**.
- ✓ Struktur **star schema** membuat query lebih cepat dan mudah dianalisis.

4.7 Kesimpulan Physical Design Data Warehouse

1. **Tujuan Physical Design**

   Physical design fokus pada **cara menyimpan, mengorganisasi, dan mengoptimalkan data** agar query analitik berjalan **cepat dan efisien**, tanpa mengubah struktur logis data.

2. **Struktur Penyimpanan**

   Data warehouse biasanya menggunakan **star schema atau snowflake schema** dan Pemilihan struktur ini mempengaruhi **join index, partitioning, dan indexing** di level fisik.

3. **Indexing**

   a. Bitmap Index: Efektif untuk kolom dengan cardinality rendah (contoh: gender, status).

   b. Join Index: Menyimpan hasil join fakta & dimensi sehingga query star lebih cepat.

   c. Clustered/Non-clustered Index: Digunakan untuk mempercepat pencarian baris tertentu di tabel.

4. **Partisi dan Kompresi**

   a. Partitioning: Memecah tabel fakta besar menjadi beberapa partisi berdasarkan waktu atau kategori untuk mempercepat query.

b. Data Compression: Mengurangi ukuran fisik tabel dan meningkatkan I/O efficiency.

5. **Materialized Views / Indexed Views**
   a. Menyimpan hasil agregasi atau join di level fisik untuk mempercepat query.
   b. Biasanya digunakan untuk star query atau laporan rutin.
6. **Pertimbangan Kinerja**
   a. Optimasi query analitik menjadi fokus utama, bukan transaksi.
   b. Perlu memperhitungkan trade-off antara update cost vs read performance.
7. **Ringkasan Praktis**

   Physical design adalah jembatan antara logical design dan performa nyata di data warehouse:

   a. Memilih index yang tepat,
   b. Menentukan partisi dan kompresi,
   c. Menyimpan join/aggregate untuk mempercepat analisis,
   d. sehingga pengguna akhir bisa mendapatkan hasil query lebih cepat tanpa merubah logika bisnis.