upGrad ENTERPRISE

Working with Views and Indexes in SQL Server

By
Narasimha Rao T
Microsoft.Net FSD Trainer
Professional Development Trainer
tnrao.trainer@gmail.com

Microsoft® SQL Server®

Day-20 (05-Aug-2025)

1. VIEWS in SQL Server

What is a View?

A *View* is a virtual table based on the result of a SQL query.

CREATE VIEW

```
CREATE VIEW vw_EmployeeDetails AS
SELECT EmpID, FirstName, LastName, Department
FROM Employees
WHERE IsActive = 1;
```

Querying a View

```
SELECT * FROM vw_EmployeeDetails;
```

Benefits of Views

- **Security**: Limit data access to certain users
- **Simplicity**: Hide complex joins
- **Abstraction**: Decouple schema from application logic

ALTER VIEW

```
ALTER VIEW vw_EmployeeDetails AS
SELECT EmpID, FirstName, LastName, Department, Salary
FROM Employees
WHERE IsActive = 1;
```

DROP VIEW

```
DROP VIEW vw_EmployeeDetails;
```

Schema Binding

- Prevents underlying table modifications

```
CREATE VIEW vw_Department
WITH SCHEMABINDING
AS
SELECT DepartmentID, DepartmentName
FROM dbo.Departments;
```

Indexed Views (Brief Intro)

- **Materialized views** that store the results physically.
- Improve performance for expensive operations.
- Must meet several requirements (e.g., schema binding, determinism).

```
CREATE UNIQUE CLUSTERED INDEX idx_vwDepartment ON vw_Department(DepartmentID);
```

Introduction to T-SQL

Variables

```
DECLARE @EmpCount INT;
SET @EmpCount = 100;
SELECT @EmpCount;
```

OR

```
SELECT @EmpCount = COUNT(*) FROM Employees;
```

Control Flow

IF/ELSE

```
IF @EmpCount > 50
    PRINT 'High employee count';
ELSE
    PRINT 'Low employee count';
```

BEGIN/END

```
IF @EmpCount > 50
BEGIN
    PRINT 'High employee count';
    PRINT 'Send report to HR';
END
```

WHILE

```
DECLARE @i INT = 1;
WHILE @i <= 5
BEGIN
    PRINT @i;
    SET @i = @i + 1;
END
```

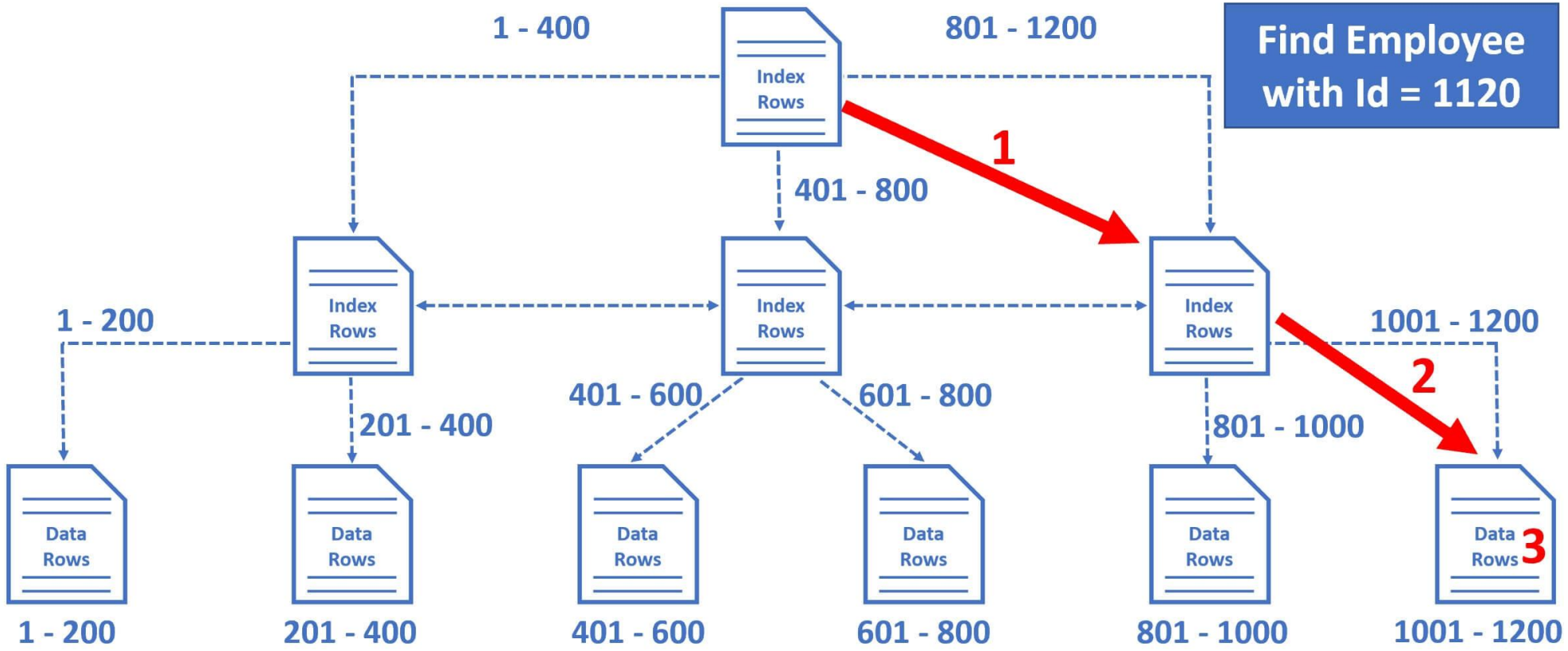
CASE

```
SELECT EmpID,
       CASE
           WHEN Salary > 50000 THEN 'High'
           WHEN Salary BETWEEN 30000 AND 50000 THEN 'Medium'
           ELSE 'Low'
       END AS SalaryLevel
FROM Employees;
```

Working with Indexes

Importance of Indexes

- Indexes **speed up data retrieval**
- Work like a book index — quickly find what you need


1 - 400
801 - 1200
Find Employee with Id = 1120
Index Rows
1
401 - 800
Index Rows
Index Rows
Index Rows
1 - 200
1001 - 1200
2
401 - 600
601 - 800
201 - 400
801 - 1000
Data Rows
Data Rows
Data Rows
Data Rows
Data Rows
Data Rows 3
1 - 200
201 - 400
401 - 600
601 - 800
801 - 1000
1001 - 1200

Types of Indexes

1. Clustered Index

- One per table, sorts table data

```
CREATE CLUSTERED INDEX idx_EmpID ON Employees(EmpID);
```

2. Non-Clustered Index

- Separate structure with pointers

```
CREATE NONCLUSTERED INDEX idx_LastName ON Employees(LastName);
```

3. Unique Index

- Ensures column(s) have unique values

```
CREATE UNIQUE INDEX idx_Email ON Employees(Email);
```

4. Composite Index

- Includes multiple columns

```
CREATE INDEX idx_NameDept ON Employees(LastName, Department);
```

Index Management

CREATE INDEX

```
CREATE INDEX idx_Salary ON Employees(Salary);
```

DROP INDEX

```
DROP INDEX idx_Salary ON Employees;
```

ALTER INDEX

Rebuild (fixes fragmentation):

```
ALTER INDEX ALL ON Employees REBUILD;
```

Reorganize (less intensive):

```
ALTER INDEX ALL ON Employees REORGANIZE;
```

Disable:

```
ALTER INDEX idx_Salary ON Employees DISABLE;
```

Index Fragmentation & Maintenance

- Fragmentation = broken storage = slower queries
- Use DMVs to check:

```
SELECT * FROM sys.dm_db_index_physical_stats(DB_ID(), NULL, NULL, NULL, 'DETAILED');
```

- Rebuild/Reorganize accordingly.

Included Columns & Covering Indexes

Covering Index: An index that contains *all* the columns required by a query.

```
CREATE NONCLUSTERED INDEX idx_Covering
ON Employees(DepartmentID)
INCLUDE (FirstName, LastName);
```

Filtered Indexes

- Index with a WHERE clause

```
CREATE NONCLUSTERED INDEX idx_ActiveEmployees
ON Employees(IsActive)
WHERE IsActive = 1;
```

Index Design Guidelines (Real-World Case Study)

✅ Do's:

- Index columns used in JOINs, WHERE, ORDER BY
- Use **covering indexes** for frequent queries
- Monitor fragmentation and maintain indexes regularly

❌ Don’ts:

- Don’t index everything—it increases write overhead
- Avoid wide composite indexes unless necessary

Case Study:

E-Commerce App:

- Use filtered index for `Orders WHERE Status = 'Pending'`
- Clustered index on `OrderID`
- Composite index on `(CustomerID, OrderDate)` for dashboard reports

upGrad ENTERPRISE

Q & A

Narasimha Rao T
tnrao.trainer@gmail.com