# Complete Guide to Indexes in SSMS

## Table of Contents

1. [What is an Index?](#what-is-an-index)
2. [Types of Indexes](#types-of-indexes)
3. [Index Structure and Storage](#index-structure-and-storage)
4. [Creating Indexes - Basic Syntax](#creating-indexes---basic-syntax)
5. [Clustered Indexes](#clustered-indexes)
6. [Non-Clustered Indexes](#non-clustered-indexes)
7. [Unique Indexes](#unique-indexes)
8. [Composite Indexes](#composite-indexes)
9. [Covering Indexes](#covering-indexes)
10. [Filtered Indexes](#filtered-indexes)
11. [Columnstore Indexes](#columnstore-indexes)
12. [Index Management](#index-management)
13. [Index Performance Analysis](#index-performance-analysis)
14. [Index Optimization Strategies](#index-optimization-strategies)
15. [Managing Indexes in SSMS](#managing-indexes-in-ssms)
16. [Best Practices](#best-practices)

## What is an Index?

An **Index** in SQL Server is a database object that improves the speed of data retrieval operations on a table. It creates shortcuts to data, similar to an index in a book, allowing the database engine to find rows quickly without scanning the entire table.

### Key Benefits:

* **Faster Query Performance**: Dramatically reduces query execution time
* **Efficient Sorting**: ORDER BY operations execute faster
* **Quick Joins**: JOIN operations between tables are optimized
* **Unique Constraint Enforcement**: Ensures data uniqueness

### Costs:

* **Storage Space**: Indexes require additional disk space
* **Maintenance Overhead**: INSERT, UPDATE, DELETE operations become slower
* **Memory Usage**: Indexes consume memory when loaded

## Types of Indexes

### 1. Clustered Index

* **One per table**: Only one clustered index per table
* **Data Storage**: Table data is physically stored in index order
* **Leaf Level**: Contains actual table data

### 2. Non-Clustered Index

* **Multiple allowed**: Up to 999 non-clustered indexes per table
* **Separate Structure**: Index structure separate from table data
* **Leaf Level**: Contains pointers to table data

### 3. Unique Index

* **Data Integrity**: Enforces uniqueness of values
* **Can be Clustered or Non-Clustered**

### 4. Composite Index

* **Multiple Columns**: Index on two or more columns
* **Column Order Matters**: First column is most important

### 5. Covering Index

* **Includes Non-Key Columns**: Additional columns for covering queries
* **Reduces Key Lookups**: All required data in index

### 6. Filtered Index

* **Conditional**: Index on subset of rows based on WHERE condition
* **Space Efficient**: Smaller than full table index

### 7. Columnstore Index

* **Columnar Storage**: Data stored by columns instead of rows
* **Analytics Optimized**: Excellent for data warehousing queries

## Index Structure and Storage

### B-Tree Structure

Root Level  
 |  
 Intermediate Levels  
 / \  
 Leaf Level Leaf Level  
 (Data Pages) (Data Pages)

### Sample Tables Setup

-- Create sample tables for demonstrations  
CREATE TABLE employees (  
 employee\_id INT IDENTITY(1,1),  
 first\_name VARCHAR(50),  
 last\_name VARCHAR(50),  
 email VARCHAR(100),  
 phone VARCHAR(20),  
 department VARCHAR(50),  
 position VARCHAR(50),  
 salary DECIMAL(10,2),  
 hire\_date DATE,  
 manager\_id INT,  
 is\_active BIT DEFAULT 1,  
 last\_updated DATETIME DEFAULT GETDATE()  
);  
  
CREATE TABLE orders (  
 order\_id INT IDENTITY(1,1),  
 customer\_id INT,  
 employee\_id INT,  
 order\_date DATETIME,  
 ship\_date DATETIME,  
 total\_amount DECIMAL(10,2),  
 status VARCHAR(20),  
 region VARCHAR(50)  
);  
  
CREATE TABLE order\_details (  
 detail\_id INT IDENTITY(1,1),  
 order\_id INT,  
 product\_id INT,  
 quantity INT,  
 unit\_price DECIMAL(8,2),  
 discount DECIMAL(3,2)  
);  
  
-- Insert sample data  
INSERT INTO employees VALUES  
('John', 'Doe', 'john.doe@company.com', '555-1001', 'IT', 'Developer', 75000, '2020-01-15', NULL, 1, GETDATE()),  
('Jane', 'Smith', 'jane.smith@company.com', '555-1002', 'IT', 'Senior Developer', 85000, '2019-03-10', 1, 1, GETDATE()),  
('Bob', 'Johnson', 'bob.johnson@company.com', '555-1003', 'Sales', 'Sales Rep', 55000, '2021-06-20', NULL, 1, GETDATE()),  
('Alice', 'Brown', 'alice.brown@company.com', '555-1004', 'HR', 'HR Manager', 70000, '2018-09-05', NULL, 1, GETDATE());  
  
-- Generate more sample data for demonstration  
DECLARE @i INT = 1;  
WHILE @i <= 1000  
BEGIN  
 INSERT INTO orders VALUES  
 (@i % 100 + 1, @i % 4 + 1, DATEADD(day, -@i, GETDATE()),   
 DATEADD(day, -@i + 3, GETDATE()), (@i \* 123.45) % 10000,   
 CASE @i % 4 WHEN 0 THEN 'Completed' WHEN 1 THEN 'Pending' WHEN 2 THEN 'Shipped' ELSE 'Cancelled' END,  
 CASE @i % 3 WHEN 0 THEN 'North' WHEN 1 THEN 'South' ELSE 'West' END);  
 SET @i = @i + 1;  
END;

## Creating Indexes - Basic Syntax

### Basic Index Creation

-- Create Non-Clustered Index  
CREATE INDEX IX\_IndexName ON TableName (ColumnName);  
  
-- Create Unique Index  
CREATE UNIQUE INDEX IX\_IndexName ON TableName (ColumnName);  
  
-- Create Clustered Index  
CREATE CLUSTERED INDEX IX\_IndexName ON TableName (ColumnName);  
  
-- Create Composite Index  
CREATE INDEX IX\_IndexName ON TableName (Column1, Column2, Column3);  
  
-- Drop Index  
DROP INDEX IX\_IndexName ON TableName;  
  
-- Disable Index  
ALTER INDEX IX\_IndexName ON TableName DISABLE;  
  
-- Rebuild Index  
ALTER INDEX IX\_IndexName ON TableName REBUILD;  
  
-- Reorganize Index  
ALTER INDEX IX\_IndexName ON TableName REORGANIZE;

## Clustered Indexes

### Primary Key Clustered Index (Automatic)

-- When creating primary key, clustered index is created automatically  
ALTER TABLE employees   
ADD CONSTRAINT PK\_employees PRIMARY KEY CLUSTERED (employee\_id);  
  
-- View the clustered index  
SELECT   
 i.name AS index\_name,  
 i.type\_desc,  
 i.is\_unique,  
 i.is\_primary\_key,  
 c.name AS column\_name  
FROM sys.indexes i  
JOIN sys.index\_columns ic ON i.object\_id = ic.object\_id AND i.index\_id = ic.index\_id  
JOIN sys.columns c ON ic.object\_id = c.object\_id AND ic.column\_id = c.column\_id  
WHERE i.object\_id = OBJECT\_ID('employees')  
AND i.type\_desc = 'CLUSTERED';

### Custom Clustered Index

-- Create table without primary key first  
CREATE TABLE sales\_data (  
 sale\_id INT,  
 sale\_date DATE,  
 amount DECIMAL(10,2),  
 region VARCHAR(50)  
);  
  
-- Create clustered index on date column (for time-series data)  
CREATE CLUSTERED INDEX IX\_sales\_data\_date ON sales\_data (sale\_date, region);  
  
-- Insert data to see physical ordering  
INSERT INTO sales\_data VALUES  
(1, '2023-01-15', 1000.00, 'North'),  
(3, '2023-01-10', 1500.00, 'South'),  
(2, '2023-01-20', 1200.00, 'East'),  
(5, '2023-01-05', 800.00, 'West'),  
(4, '2023-01-25', 2000.00, 'North');  
  
-- Data will be physically stored ordered by sale\_date, then region  
SELECT \* FROM sales\_data; -- Notice the physical order

### Clustered Index Performance Benefits

-- Query that benefits from clustered index  
SELECT \* FROM sales\_data   
WHERE sale\_date BETWEEN '2023-01-10' AND '2023-01-20'  
ORDER BY sale\_date;  
  
-- Range queries are very efficient with clustered indexes  
SELECT \* FROM employees   
WHERE employee\_id BETWEEN 100 AND 200;

## Non-Clustered Indexes

### Single Column Indexes

-- Index for frequent WHERE clause searches  
CREATE INDEX IX\_employees\_last\_name ON employees (last\_name);  
  
-- Index for JOIN operations  
CREATE INDEX IX\_orders\_employee\_id ON orders (employee\_id);  
  
-- Index for date range queries  
CREATE INDEX IX\_orders\_order\_date ON orders (order\_date);  
  
-- Index for department searches  
CREATE INDEX IX\_employees\_department ON employees (department);

### Performance Comparison

-- Query without index (table scan)  
SELECT \* FROM employees WHERE last\_name = 'Johnson';  
  
-- After creating index IX\_employees\_last\_name, same query uses index seek  
-- Check execution plan to see the difference  
  
-- View index usage statistics  
SELECT   
 i.name AS index\_name,  
 s.user\_seeks,  
 s.user\_scans,  
 s.user\_lookups,  
 s.user\_updates,  
 s.last\_user\_seek,  
 s.last\_user\_scan  
FROM sys.indexes i  
LEFT JOIN sys.dm\_db\_index\_usage\_stats s   
 ON i.object\_id = s.object\_id AND i.index\_id = s.index\_id  
WHERE i.object\_id = OBJECT\_ID('employees');

## Unique Indexes

### Enforcing Data Uniqueness

-- Create unique index on email column  
CREATE UNIQUE INDEX IX\_employees\_email ON employees (email);  
  
-- Try to insert duplicate email (will fail)  
-- INSERT INTO employees VALUES ('Test', 'User', 'john.doe@company.com', ...);  
  
-- Create unique composite index  
CREATE UNIQUE INDEX IX\_employees\_name\_dept ON employees (first\_name, last\_name, department);  
  
-- Unique index with NULL handling  
CREATE UNIQUE INDEX IX\_employees\_phone ON employees (phone)   
WHERE phone IS NOT NULL; -- Filtered unique index

### Unique Index vs Unique Constraint

-- Using UNIQUE constraint (creates unique index automatically)  
ALTER TABLE employees   
ADD CONSTRAINT UQ\_employees\_email UNIQUE (email);  
  
-- View constraint and associated index  
SELECT   
 kc.name AS constraint\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 c.name AS column\_name  
FROM sys.key\_constraints kc  
JOIN sys.indexes i ON kc.parent\_object\_id = i.object\_id AND kc.unique\_index\_id = i.index\_id  
JOIN sys.index\_columns ic ON i.object\_id = ic.object\_id AND i.index\_id = ic.index\_id  
JOIN sys.columns c ON ic.object\_id = c.object\_id AND ic.column\_id = c.column\_id  
WHERE kc.parent\_object\_id = OBJECT\_ID('employees');

## Composite Indexes

### Multi-Column Index Creation

-- Composite index for complex WHERE clauses  
CREATE INDEX IX\_orders\_customer\_date ON orders (customer\_id, order\_date);  
  
-- Index for GROUP BY operations  
CREATE INDEX IX\_orders\_region\_status ON orders (region, status);  
  
-- Index with different sort orders  
CREATE INDEX IX\_employees\_dept\_salary ON employees (department ASC, salary DESC);

### Column Order Importance

-- Index: (department, salary, hire\_date)  
CREATE INDEX IX\_employees\_dept\_sal\_date ON employees (department, salary, hire\_date);  
  
-- These queries can use the index effectively:  
-- 1. Uses all columns  
SELECT \* FROM employees   
WHERE department = 'IT' AND salary > 70000 AND hire\_date > '2020-01-01';  
  
-- 2. Uses leading columns (department, salary)  
SELECT \* FROM employees   
WHERE department = 'IT' AND salary > 70000;  
  
-- 3. Uses first column only  
SELECT \* FROM employees WHERE department = 'IT';  
  
-- This query CANNOT use the index effectively (salary without department):  
SELECT \* FROM employees WHERE salary > 70000; -- May still use index but less efficiently  
  
-- This query CANNOT use the index effectively (non-leading column only):  
SELECT \* FROM employees WHERE hire\_date > '2020-01-01';

### Composite Index Best Practices

-- Order columns by selectivity (most selective first)  
CREATE INDEX IX\_orders\_optimal ON orders (  
 customer\_id, -- Most selective (specific customer)  
 status, -- Moderately selective (few status values)  
 region -- Least selective (only 3 regions)  
);  
  
-- Include frequently used sort columns  
CREATE INDEX IX\_employees\_search\_sort ON employees (  
 department, -- WHERE clause  
 last\_name, -- WHERE clause  
 salary -- ORDER BY clause  
);

## Covering Indexes

### Index with INCLUDE Columns

-- Covering index to avoid key lookups  
CREATE INDEX IX\_employees\_dept\_covering ON employees (department)  
INCLUDE (first\_name, last\_name, salary, hire\_date);  
  
-- This query is completely covered by the index (no key lookup needed)  
SELECT first\_name, last\_name, salary, hire\_date  
FROM employees  
WHERE department = 'IT';  
  
-- Compare with non-covering index  
CREATE INDEX IX\_employees\_dept\_only ON employees (department);  
  
-- Same query now requires key lookup for additional columns  
SELECT first\_name, last\_name, salary, hire\_date  
FROM employees  
WHERE department = 'Sales';

### Covering Index for JOIN Operations

-- Covering index for order-employee joins  
CREATE INDEX IX\_orders\_emp\_covering ON orders (employee\_id)  
INCLUDE (order\_date, total\_amount, status);  
  
-- Covering index for employee side of join  
CREATE INDEX IX\_employees\_join\_covering ON employees (employee\_id)  
INCLUDE (first\_name, last\_name, department);  
  
-- Query that benefits from covering indexes  
SELECT   
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 e.department,  
 o.order\_date,  
 o.total\_amount,  
 o.status  
FROM employees e  
JOIN orders o ON e.employee\_id = o.employee\_id  
WHERE e.department = 'Sales'  
AND o.order\_date >= '2023-01-01';

## Filtered Indexes

### Indexes on Subsets of Data

-- Index only on active employees  
CREATE INDEX IX\_employees\_active\_dept ON employees (department)  
WHERE is\_active = 1;  
  
-- Index only on recent orders  
CREATE INDEX IX\_orders\_recent ON orders (order\_date, customer\_id)  
WHERE order\_date >= '2023-01-01';  
  
-- Index only on non-null values  
CREATE INDEX IX\_employees\_manager ON employees (manager\_id)  
WHERE manager\_id IS NOT NULL;  
  
-- Index only on high-value orders  
CREATE INDEX IX\_orders\_high\_value ON orders (order\_date)  
INCLUDE (customer\_id, total\_amount)  
WHERE total\_amount > 1000;

### Filtered Index Benefits

-- Statistics show filtered index efficiency  
SELECT   
 i.name AS index\_name,  
 i.type\_desc,  
 i.has\_filter,  
 i.filter\_definition,  
 p.rows AS index\_rows,  
 p.data\_compression\_desc  
FROM sys.indexes i  
JOIN sys.partitions p ON i.object\_id = p.object\_id AND i.index\_id = p.index\_id  
WHERE i.object\_id = OBJECT\_ID('employees')  
AND i.has\_filter = 1;  
  
-- Query that benefits from filtered index  
SELECT first\_name, last\_name, department  
FROM employees  
WHERE is\_active = 1 AND department = 'IT';

## Columnstore Indexes

### Clustered Columnstore Index

-- Create table for analytics workload  
CREATE TABLE sales\_fact (  
 sale\_id INT,  
 product\_id INT,  
 customer\_id INT,  
 sale\_date DATE,  
 quantity INT,  
 unit\_price DECIMAL(8,2),  
 total\_amount DECIMAL(10,2),  
 region VARCHAR(50)  
);  
  
-- Create clustered columnstore index  
CREATE CLUSTERED COLUMNSTORE INDEX CCI\_sales\_fact ON sales\_fact;  
  
-- Insert sample data  
INSERT INTO sales\_fact VALUES  
(1, 101, 1001, '2023-01-01', 5, 19.99, 99.95, 'North'),  
(2, 102, 1002, '2023-01-02', 3, 29.99, 89.97, 'South'),  
(3, 103, 1003, '2023-01-03', 2, 49.99, 99.98, 'East');

### Non-Clustered Columnstore Index

-- Add columnstore index to existing table  
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCI\_orders\_analytics   
ON orders (customer\_id, order\_date, total\_amount, region);  
  
-- Queries that benefit from columnstore  
-- Aggregation queries  
SELECT   
 region,  
 YEAR(order\_date) AS order\_year,  
 COUNT(\*) AS order\_count,  
 SUM(total\_amount) AS total\_sales,  
 AVG(total\_amount) AS avg\_order\_value  
FROM orders  
GROUP BY region, YEAR(order\_date);  
  
-- Large range scans  
SELECT COUNT(\*)  
FROM orders  
WHERE order\_date BETWEEN '2022-01-01' AND '2023-12-31'  
AND total\_amount > 500;

## Index Management

### Rebuilding Indexes

-- Rebuild single index  
ALTER INDEX IX\_employees\_last\_name ON employees REBUILD;  
  
-- Rebuild all indexes on table  
ALTER INDEX ALL ON employees REBUILD;  
  
-- Rebuild with options  
ALTER INDEX IX\_employees\_last\_name ON employees REBUILD  
WITH (  
 FILLFACTOR = 80,  
 ONLINE = ON,  
 MAXDOP = 4  
);  
  
-- Rebuild columnstore index  
ALTER INDEX CCI\_sales\_fact ON sales\_fact REBUILD;

### Reorganizing Indexes

-- Reorganize single index (online operation)  
ALTER INDEX IX\_employees\_last\_name ON employees REORGANIZE;  
  
-- Reorganize all indexes  
ALTER INDEX ALL ON employees REORGANIZE;  
  
-- Reorganize with LOB compaction  
ALTER INDEX IX\_employees\_last\_name ON employees REORGANIZE  
WITH (LOB\_COMPACTION = ON);

### Index Fragmentation Analysis

-- Check index fragmentation  
SELECT   
 i.name AS index\_name,  
 ips.index\_type\_desc,  
 ips.avg\_fragmentation\_in\_percent,  
 ips.fragment\_count,  
 ips.page\_count,  
 CASE   
 WHEN ips.avg\_fragmentation\_in\_percent < 10 THEN 'No Action Needed'  
 WHEN ips.avg\_fragmentation\_in\_percent < 30 THEN 'Reorganize'  
 ELSE 'Rebuild'  
 END AS recommended\_action  
FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), OBJECT\_ID('employees'), NULL, NULL, 'DETAILED') ips  
JOIN sys.indexes i ON ips.object\_id = i.object\_id AND ips.index\_id = i.index\_id  
WHERE i.name IS NOT NULL  
ORDER BY ips.avg\_fragmentation\_in\_percent DESC;

### Automated Index Maintenance

-- Stored procedure for index maintenance  
CREATE PROCEDURE sp\_IndexMaintenance  
 @table\_name VARCHAR(128) = NULL,  
 @fragmentation\_threshold\_reorganize FLOAT = 10.0,  
 @fragmentation\_threshold\_rebuild FLOAT = 30.0  
AS  
BEGIN  
 DECLARE @sql NVARCHAR(MAX);  
 DECLARE @index\_name VARCHAR(128), @fragmentation FLOAT, @object\_id INT, @index\_id INT;  
   
 DECLARE index\_cursor CURSOR FOR  
 SELECT   
 i.name,  
 ips.avg\_fragmentation\_in\_percent,  
 ips.object\_id,  
 ips.index\_id  
 FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), OBJECT\_ID(@table\_name), NULL, NULL, 'DETAILED') ips  
 JOIN sys.indexes i ON ips.object\_id = i.object\_id AND ips.index\_id = i.index\_id  
 WHERE i.name IS NOT NULL  
 AND ips.avg\_fragmentation\_in\_percent > @fragmentation\_threshold\_reorganize;  
   
 OPEN index\_cursor;  
 FETCH NEXT FROM index\_cursor INTO @index\_name, @fragmentation, @object\_id, @index\_id;  
   
 WHILE @@FETCH\_STATUS = 0  
 BEGIN  
 IF @fragmentation >= @fragmentation\_threshold\_rebuild  
 BEGIN  
 SET @sql = 'ALTER INDEX ' + QUOTENAME(@index\_name) + ' ON ' + OBJECT\_NAME(@object\_id) + ' REBUILD';  
 PRINT 'Rebuilding: ' + @sql;  
 END  
 ELSE  
 BEGIN  
 SET @sql = 'ALTER INDEX ' + QUOTENAME(@index\_name) + ' ON ' + OBJECT\_NAME(@object\_id) + ' REORGANIZE';  
 PRINT 'Reorganizing: ' + @sql;  
 END  
   
 EXEC sp\_executesql @sql;  
   
 FETCH NEXT FROM index\_cursor INTO @index\_name, @fragmentation, @object\_id, @index\_id;  
 END  
   
 CLOSE index\_cursor;  
 DEALLOCATE index\_cursor;  
END  
  
-- Execute maintenance  
EXEC sp\_IndexMaintenance @table\_name = 'orders';

## Index Performance Analysis

### Index Usage Statistics

-- View index usage statistics  
SELECT   
 OBJECT\_NAME(i.object\_id) AS table\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 s.user\_seeks,  
 s.user\_scans,  
 s.user\_lookups,  
 s.user\_updates,  
 s.last\_user\_seek,  
 s.last\_user\_scan,  
 s.last\_user\_lookup,  
 s.last\_user\_update,  
 CASE   
 WHEN s.user\_seeks + s.user\_scans + s.user\_lookups = 0 THEN 'Unused'  
 WHEN s.user\_updates > (s.user\_seeks + s.user\_scans + s.user\_lookups) \* 2 THEN 'High Maintenance'  
 ELSE 'Good Usage'  
 END AS usage\_pattern  
FROM sys.indexes i  
LEFT JOIN sys.dm\_db\_index\_usage\_stats s   
 ON i.object\_id = s.object\_id AND i.index\_id = s.index\_id AND s.database\_id = DB\_ID()  
WHERE i.object\_id = OBJECT\_ID('employees')  
ORDER BY s.user\_seeks + s.user\_scans + s.user\_lookups DESC;

### Missing Index Suggestions

-- Find missing index suggestions  
SELECT   
 d.statement AS table\_name,  
 d.equality\_columns,  
 d.inequality\_columns,  
 d.included\_columns,  
 s.user\_seeks,  
 s.user\_scans,  
 s.last\_user\_seek,  
 s.avg\_total\_user\_cost,  
 s.avg\_user\_impact,  
 'CREATE INDEX IX\_' +   
 REPLACE(REPLACE(REPLACE(d.statement, '[', ''), ']', ''), '.', '\_') +   
 '\_Missing ON ' + d.statement +   
 ' (' + ISNULL(d.equality\_columns, '') +   
 CASE WHEN d.inequality\_columns IS NOT NULL THEN   
 CASE WHEN d.equality\_columns IS NOT NULL THEN ',' ELSE '' END + d.inequality\_columns   
 ELSE '' END + ')' +  
 CASE WHEN d.included\_columns IS NOT NULL THEN ' INCLUDE (' + d.included\_columns + ')' ELSE '' END  
 AS create\_statement  
FROM sys.dm\_db\_missing\_index\_details d  
JOIN sys.dm\_db\_missing\_index\_groups g ON d.index\_handle = g.index\_handle  
JOIN sys.dm\_db\_missing\_index\_group\_stats s ON g.index\_group\_handle = s.group\_handle  
WHERE d.database\_id = DB\_ID()  
ORDER BY s.avg\_total\_user\_cost \* s.avg\_user\_impact \* (s.user\_seeks + s.user\_scans) DESC;

### Index Size and Space Usage

-- Analyze index sizes  
SELECT   
 OBJECT\_NAME(i.object\_id) AS table\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 i.is\_unique,  
 i.fill\_factor,  
 ps.in\_row\_data\_page\_count,  
 ps.in\_row\_used\_page\_count,  
 ps.row\_count,  
 CAST(ps.in\_row\_used\_page\_count \* 8.0 / 1024 AS DECIMAL(10,2)) AS index\_size\_mb,  
 CAST(ps.in\_row\_data\_page\_count \* 8.0 / 1024 AS DECIMAL(10,2)) AS allocated\_size\_mb  
FROM sys.indexes i  
JOIN sys.dm\_db\_partition\_stats ps ON i.object\_id = ps.object\_id AND i.index\_id = ps.index\_id  
WHERE i.object\_id IN (OBJECT\_ID('employees'), OBJECT\_ID('orders'))  
ORDER BY ps.in\_row\_used\_page\_count DESC;

## Index Optimization Strategies

### Query-Specific Index Design

-- Analyze a specific query's execution plan  
-- Query 1: Employee search with sorting  
SELECT employee\_id, first\_name, last\_name, salary  
FROM employees  
WHERE department = 'IT' AND salary > 60000  
ORDER BY last\_name, first\_name;  
  
-- Optimal index for this query  
CREATE INDEX IX\_employees\_query1\_optimal ON employees (department, salary)  
INCLUDE (first\_name, last\_name);  
  
-- Query 2: Complex join with filtering  
SELECT   
 e.first\_name + ' ' + e.last\_name AS employee\_name,  
 o.order\_date,  
 o.total\_amount  
FROM employees e  
JOIN orders o ON e.employee\_id = o.employee\_id  
WHERE e.department = 'Sales'  
AND o.order\_date >= '2023-01-01'  
AND o.total\_amount > 500  
ORDER BY o.order\_date DESC;  
  
-- Optimal indexes for this query  
CREATE INDEX IX\_employees\_sales\_join ON employees (employee\_id)  
INCLUDE (first\_name, last\_name)  
WHERE department = 'Sales';  
  
CREATE INDEX IX\_orders\_date\_amount ON orders (employee\_id, order\_date, total\_amount)  
WHERE order\_date >= '2023-01-01' AND total\_amount > 500;

### Index Consolidation

-- Instead of multiple single-column indexes:  
-- CREATE INDEX IX\_orders\_customer ON orders (customer\_id);  
-- CREATE INDEX IX\_orders\_date ON orders (order\_date);  
-- CREATE INDEX IX\_orders\_status ON orders (status);  
  
-- Create one composite index that can serve multiple queries:  
CREATE INDEX IX\_orders\_consolidated ON orders (customer\_id, order\_date, status)  
INCLUDE (total\_amount, region);  
  
-- This index can efficiently support:  
-- 1. WHERE customer\_id = X  
-- 2. WHERE customer\_id = X AND order\_date = Y  
-- 3. WHERE customer\_id = X AND order\_date = Y AND status = Z  
-- 4. All with covered columns for SELECT lists

### Partitioned Index Strategy

-- Create partitioned table and indexes  
CREATE PARTITION FUNCTION pf\_order\_date (DATE)  
AS RANGE RIGHT FOR VALUES ('2023-01-01', '2023-04-01', '2023-07-01', '2023-10-01');  
  
CREATE PARTITION SCHEME ps\_order\_date  
AS PARTITION pf\_order\_date ALL TO ([PRIMARY]);  
  
-- Create partitioned table  
CREATE TABLE orders\_partitioned (  
 order\_id INT IDENTITY(1,1),  
 customer\_id INT,  
 employee\_id INT,  
 order\_date DATE,  
 ship\_date DATE,  
 total\_amount DECIMAL(10,2),  
 status VARCHAR(20),  
 region VARCHAR(50)  
) ON ps\_order\_date (order\_date);  
  
-- Create aligned partitioned index  
CREATE INDEX IX\_orders\_part\_customer ON orders\_partitioned (customer\_id, order\_date)  
ON ps\_order\_date (order\_date);

## Managing Indexes in SSMS

### Using SSMS Interface

#### Creating Indexes through GUI:

1. **Right-click table** → “Design”
2. **Right-click table** → “Indexes/Keys”
3. **Expand table** → Right-click “Indexes” → “New Index”

#### Index Properties in SSMS:

-- View index properties programmatically  
SELECT   
 i.name AS index\_name,  
 i.type\_desc,  
 i.is\_unique,  
 i.is\_primary\_key,  
 i.is\_unique\_constraint,  
 i.fill\_factor,  
 i.ignore\_dup\_key,  
 i.allow\_row\_locks,  
 i.allow\_page\_locks,  
 i.has\_filter,  
 i.filter\_definition  
FROM sys.indexes i  
WHERE i.object\_id = OBJECT\_ID('employees')  
AND i.name IS NOT NULL;

### Index Monitoring Queries

-- Complete index information for a table  
SELECT   
 t.name AS table\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 i.is\_unique,  
 i.is\_primary\_key,  
 STRING\_AGG(c.name, ', ') WITHIN GROUP (ORDER BY ic.key\_ordinal) AS key\_columns,  
 i.has\_filter,  
 i.filter\_definition,  
 ps.row\_count,  
 CAST(ps.in\_row\_used\_page\_count \* 8.0 / 1024 AS DECIMAL(10,2)) AS size\_mb  
FROM sys.tables t  
JOIN sys.indexes i ON t.object\_id = i.object\_id  
JOIN sys.index\_columns ic ON i.object\_id = ic.object\_id AND i.index\_id = ic.index\_id  
JOIN sys.columns c ON ic.object\_id = c.object\_id AND ic.column\_id = c.column\_id  
JOIN sys.dm\_db\_partition\_stats ps ON i.object\_id = ps.object\_id AND i.index\_id = ps.index\_id  
WHERE t.name = 'employees' AND i.name IS NOT NULL  
GROUP BY t.name, i.name, i.type\_desc, i.is\_unique, i.is\_primary\_key,   
 i.has\_filter, i.filter\_definition, ps.row\_count, ps.in\_row\_used\_page\_count  
ORDER BY i.type\_desc, i.name;

## Best Practices

### Index Design Guidelines

#### 1. Primary Key Selection

-- Good: Use identity column for primary key  
CREATE TABLE good\_table (  
 id INT IDENTITY(1,1) PRIMARY KEY, -- Clustered index  
 data VARCHAR(100)  
);  
  
-- Avoid: Wide or frequently changing primary key  
CREATE TABLE avoid\_table (  
 natural\_key VARCHAR(50) PRIMARY KEY, -- May cause fragmentation  
 data VARCHAR(100)  
);

#### 2. Index Column Selection

-- Good: Index on selective columns  
CREATE INDEX IX\_employees\_email ON employees (email); -- High selectivity  
CREATE INDEX IX\_employees\_dept\_salary ON employees (department, salary); -- Moderate selectivity  
  
-- Avoid: Index on non-selective columns  
-- CREATE INDEX IX\_employees\_gender ON employees (gender); -- Low selectivity (M/F)  
-- CREATE INDEX IX\_employees\_active ON employees (is\_active); -- Low selectivity (0/1)  
  
-- Better: Use filtered indexes for low-selectivity columns  
CREATE INDEX IX\_employees\_inactive ON employees (employee\_id)   
WHERE is\_active = 0; -- Only index inactive employees

#### 3. Composite Index Order

-- Order by: Equality first, then Inequality, then ORDER BY columns  
CREATE INDEX IX\_orders\_optimal\_order ON orders (  
 status, -- Equality condition (WHERE status = 'Completed')  
 customer\_id, -- Equality condition (WHERE customer\_id = 123)  
 order\_date, -- Range condition (WHERE order\_date >= '2023-01-01')  
 total\_amount -- ORDER BY clause  
);  
  
-- Include frequently accessed columns  
CREATE INDEX IX\_orders\_covering ON orders (status, customer\_id)  
INCLUDE (order\_date, total\_amount, region);

#### 4. Maintenance Considerations

-- Set appropriate fill factor for frequently updated tables  
CREATE INDEX IX\_orders\_updated\_frequently ON orders (order\_date)  
WITH (FILLFACTOR = 80); -- Leave 20% free space for updates  
  
-- For read-only or rarely updated tables  
CREATE INDEX IX\_archive\_data ON archive\_table (date\_column)  
WITH (FILLFACTOR = 100); -- No free space needed

### Index Naming Conventions

-- Consistent naming convention  
-- IX\_TableName\_ColumnName(s)  
CREATE INDEX IX\_employees\_last\_name ON employees (last\_name);  
CREATE INDEX IX\_employees\_dept\_salary ON employees (department, salary);  
CREATE INDEX IX\_orders\_customer\_date ON orders (customer\_id, order\_date);  
  
-- For unique indexes  
-- UX\_TableName\_ColumnName(s)  
CREATE UNIQUE INDEX UX\_employees\_email ON employees (email);  
  
-- For filtered indexes  
-- IX\_TableName\_ColumnName\_FilterDesc  
CREATE INDEX IX\_employees\_salary\_active ON employees (salary)  
WHERE is\_active = 1;

### Performance Monitoring

-- Regular index health check procedure  
CREATE PROCEDURE sp\_IndexHealthCheck  
AS  
BEGIN  
 -- 1. Index fragmentation  
 SELECT   
 OBJECT\_NAME(ips.object\_id) AS table\_name,  
 i.name AS index\_name,  
 ips.avg\_fragmentation\_in\_percent,  
 ips.page\_count,  
 CASE   
 WHEN ips.avg\_fragmentation\_in\_percent < 10 THEN 'Healthy'  
 WHEN ips.avg\_fragmentation\_in\_percent < 30 THEN 'Needs Reorganization'  
 ELSE 'Needs Rebuild'  
 END AS health\_status  
 FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), NULL, NULL, NULL, 'SAMPLED') ips  
 JOIN sys.indexes i ON ips.object\_id = i.object\_id AND ips.index\_id = i.index\_id  
 WHERE i.name IS NOT NULL AND ips.page\_count > 100  
 ORDER BY ips.avg\_fragmentation\_in\_percent DESC;  
   
 -- 2. Unused indexes  
 SELECT   
 OBJECT\_NAME(i.object\_id) AS table\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 COALESCE(s.user\_seeks + s.user\_scans + s.user\_lookups, 0) AS total\_reads,  
 COALESCE(s.user\_updates, 0) AS total\_writes,  
 CASE   
 WHEN s.user\_seeks + s.user\_scans + s.user\_lookups IS NULL THEN 'Never Used'  
 WHEN s.user\_updates > (s.user\_seeks + s.user\_scans + s.user\_lookups) \* 5 THEN 'High Maintenance Cost'  
 ELSE 'Good Usage'  
 END AS usage\_assessment  
 FROM sys.indexes i  
 LEFT JOIN sys.dm\_db\_index\_usage\_stats s   
 ON i.object\_id = s.object\_id AND i.index\_id = s.index\_id AND s.database\_id = DB\_ID()  
 WHERE i.name IS NOT NULL AND i.is\_primary\_key = 0  
 ORDER BY total\_reads;  
END  
  
-- Execute health check  
EXEC sp\_IndexHealthCheck;

## Advanced Index Scenarios

### Index for Data Warehousing

-- Create fact table with appropriate indexes  
CREATE TABLE sales\_fact (  
 fact\_id BIGINT IDENTITY(1,1) PRIMARY KEY,  
 date\_key INT,  
 product\_key INT,  
 customer\_key INT,  
 employee\_key INT,  
 quantity INT,  
 unit\_price DECIMAL(10,2),  
 total\_amount DECIMAL(12,2),  
 cost DECIMAL(12,2),  
 profit DECIMAL(12,2)  
);  
  
-- Columnstore index for analytical queries  
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCI\_sales\_fact\_analytics   
ON sales\_fact (date\_key, product\_key, customer\_key, quantity, total\_amount, profit);  
  
-- Traditional B-tree indexes for operational queries  
CREATE INDEX IX\_sales\_fact\_date ON sales\_fact (date\_key, customer\_key)  
INCLUDE (total\_amount, profit);  
  
CREATE INDEX IX\_sales\_fact\_product ON sales\_fact (product\_key, date\_key)  
INCLUDE (quantity, total\_amount);  
  
-- Partitioned index for time-series data  
CREATE PARTITION FUNCTION pf\_sales\_date (INT)  
AS RANGE RIGHT FOR VALUES (20230101, 20230401, 20230701, 20231001);  
  
CREATE PARTITION SCHEME ps\_sales\_date  
AS PARTITION pf\_sales\_date ALL TO ([PRIMARY]);  
  
-- Create partitioned clustered index  
CREATE CLUSTERED INDEX CIX\_sales\_fact\_partitioned   
ON sales\_fact (date\_key, fact\_id)  
ON ps\_sales\_date (date\_key);

### Index for JSON Data

-- Table with JSON column  
CREATE TABLE user\_profiles (  
 user\_id INT IDENTITY(1,1) PRIMARY KEY,  
 username VARCHAR(50),  
 profile\_data NVARCHAR(MAX) CHECK (ISJSON(profile\_data) = 1),  
 created\_date DATETIME2 DEFAULT GETDATE()  
);  
  
-- Insert sample JSON data  
INSERT INTO user\_profiles VALUES  
('john\_doe', '{"age": 30, "city": "New York", "skills": ["SQL", "C#", "JavaScript"]}', GETDATE()),  
('jane\_smith', '{"age": 28, "city": "Chicago", "skills": ["Python", "R", "Machine Learning"]}', GETDATE());  
  
-- Create computed columns for JSON properties  
ALTER TABLE user\_profiles   
ADD age AS CAST(JSON\_VALUE(profile\_data, '$.age') AS INT);  
  
ALTER TABLE user\_profiles   
ADD city AS JSON\_VALUE(profile\_data, '$.city');  
  
-- Create indexes on computed columns  
CREATE INDEX IX\_user\_profiles\_age ON user\_profiles (age);  
CREATE INDEX IX\_user\_profiles\_city ON user\_profiles (city);  
  
-- Query using JSON indexes  
SELECT username, age, city  
FROM user\_profiles  
WHERE age BETWEEN 25 AND 35  
AND city = 'New York';

### Index for Temporal Tables

-- Create system-versioned temporal table  
CREATE TABLE employee\_history (  
 employee\_id INT,  
 first\_name VARCHAR(50),  
 last\_name VARCHAR(50),  
 salary DECIMAL(10,2),  
 department VARCHAR(50),  
   
 -- System columns for temporal functionality  
 valid\_from DATETIME2 GENERATED ALWAYS AS ROW START NOT NULL,  
 valid\_to DATETIME2 GENERATED ALWAYS AS ROW END NOT NULL,  
 PERIOD FOR SYSTEM\_TIME (valid\_from, valid\_to)  
)  
WITH (SYSTEM\_VERSIONING = ON (HISTORY\_TABLE = dbo.employee\_history\_archive));  
  
-- Create indexes optimized for temporal queries  
CREATE INDEX IX\_employee\_history\_time ON employee\_history (valid\_from, valid\_to, employee\_id);  
CREATE INDEX IX\_employee\_history\_archive\_time ON employee\_history\_archive (valid\_to, valid\_from, employee\_id);  
  
-- Temporal query examples  
-- Point-in-time query  
SELECT \* FROM employee\_history   
FOR SYSTEM\_TIME AS OF '2023-06-01'  
WHERE employee\_id = 1;  
  
-- Historical range query  
SELECT \* FROM employee\_history   
FOR SYSTEM\_TIME BETWEEN '2023-01-01' AND '2023-12-31'  
WHERE department = 'IT';

## Index Troubleshooting

### Common Index Problems and Solutions

#### Problem 1: High Index Fragmentation

-- Identify fragmented indexes  
SELECT   
 OBJECT\_NAME(ips.object\_id) AS table\_name,  
 i.name AS index\_name,  
 ips.avg\_fragmentation\_in\_percent,  
 ips.fragment\_count,  
 ips.page\_count,  
 'ALTER INDEX ' + QUOTENAME(i.name) + ' ON ' + QUOTENAME(OBJECT\_NAME(ips.object\_id)) +   
 CASE   
 WHEN ips.avg\_fragmentation\_in\_percent > 30 THEN ' REBUILD;'  
 WHEN ips.avg\_fragmentation\_in\_percent > 10 THEN ' REORGANIZE;'  
 ELSE ' -- No action needed'  
 END AS recommended\_action  
FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), NULL, NULL, NULL, 'DETAILED') ips  
JOIN sys.indexes i ON ips.object\_id = i.object\_id AND ips.index\_id = i.index\_id  
WHERE i.name IS NOT NULL   
AND ips.page\_count > 100  
AND ips.avg\_fragmentation\_in\_percent > 10  
ORDER BY ips.avg\_fragmentation\_in\_percent DESC;

#### Problem 2: Index Not Being Used

-- Check if indexes are being used  
WITH IndexUsage AS (  
 SELECT   
 OBJECT\_NAME(i.object\_id) AS table\_name,  
 i.name AS index\_name,  
 i.type\_desc,  
 COALESCE(s.user\_seeks, 0) + COALESCE(s.user\_scans, 0) + COALESCE(s.user\_lookups, 0) AS total\_reads,  
 COALESCE(s.user\_updates, 0) AS total\_writes,  
 s.last\_user\_seek,  
 s.last\_user\_scan,  
 CASE   
 WHEN s.user\_seeks IS NULL AND s.user\_scans IS NULL AND s.user\_lookups IS NULL THEN 'UNUSED'  
 WHEN s.user\_updates > (COALESCE(s.user\_seeks, 0) + COALESCE(s.user\_scans, 0) + COALESCE(s.user\_lookups, 0)) \* 2 THEN 'HIGH\_MAINTENANCE'  
 ELSE 'ACTIVE'  
 END AS usage\_pattern  
 FROM sys.indexes i  
 LEFT JOIN sys.dm\_db\_index\_usage\_stats s   
 ON i.object\_id = s.object\_id AND i.index\_id = s.index\_id AND s.database\_id = DB\_ID()  
 WHERE i.name IS NOT NULL  
 AND i.is\_primary\_key = 0  
 AND i.is\_unique\_constraint = 0  
)  
SELECT \*,  
 CASE usage\_pattern  
 WHEN 'UNUSED' THEN 'Consider dropping: DROP INDEX ' + QUOTENAME(index\_name) + ' ON ' + QUOTENAME(table\_name)  
 WHEN 'HIGH\_MAINTENANCE' THEN 'Review necessity - high update overhead'  
 ELSE 'Index is being used effectively'  
 END AS recommendation  
FROM IndexUsage  
ORDER BY usage\_pattern, total\_reads;

#### Problem 3: Too Many Indexes on a Table

-- Identify tables with excessive indexes  
SELECT   
 OBJECT\_NAME(i.object\_id) AS table\_name,  
 COUNT(\*) AS index\_count,  
 SUM(CASE WHEN i.type\_desc = 'CLUSTERED' THEN 1 ELSE 0 END) AS clustered\_count,  
 SUM(CASE WHEN i.type\_desc = 'NONCLUSTERED' THEN 1 ELSE 0 END) AS nonclustered\_count,  
 SUM(CASE WHEN i.type\_desc LIKE '%COLUMNSTORE%' THEN 1 ELSE 0 END) AS columnstore\_count,  
 CASE   
 WHEN COUNT(\*) > 10 THEN 'Consider index consolidation'  
 WHEN COUNT(\*) > 5 THEN 'Monitor performance'  
 ELSE 'Acceptable'  
 END AS assessment  
FROM sys.indexes i  
WHERE i.name IS NOT NULL  
GROUP BY i.object\_id  
HAVING COUNT(\*) > 5  
ORDER BY COUNT(\*) DESC;

### Index Performance Tuning

-- Create procedure to analyze query performance with indexes  
CREATE PROCEDURE sp\_AnalyzeQueryPerformance  
 @query NVARCHAR(MAX)  
AS  
BEGIN  
 -- Enable statistics  
 SET STATISTICS IO ON;  
 SET STATISTICS TIME ON;  
   
 PRINT 'Executing query with current indexes...';  
 PRINT @query;  
   
 -- Execute the query  
 EXEC sp\_executesql @query;  
   
 -- Get execution plan information  
 SELECT   
 qs.sql\_handle,  
 qs.plan\_handle,  
 qs.total\_logical\_reads,  
 qs.total\_physical\_reads,  
 qs.total\_elapsed\_time / 1000 AS total\_elapsed\_time\_ms,  
 qs.execution\_count,  
 qs.total\_logical\_reads / qs.execution\_count AS avg\_logical\_reads,  
 SUBSTRING(qt.text, (qs.statement\_start\_offset/2)+1,  
 ((CASE qs.statement\_end\_offset  
 WHEN -1 THEN DATALENGTH(qt.text)  
 ELSE qs.statement\_end\_offset  
 END - qs.statement\_start\_offset)/2) + 1) AS statement\_text  
 FROM sys.dm\_exec\_query\_stats qs  
 CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) qt  
 WHERE qt.text LIKE '%' + REPLACE(@query, '''', '''''') + '%'  
 ORDER BY qs.total\_logical\_reads DESC;  
   
 SET STATISTICS IO OFF;  
 SET STATISTICS TIME OFF;  
END  
  
-- Example usage  
EXEC sp\_AnalyzeQueryPerformance   
 @query = N'SELECT \* FROM employees WHERE department = ''IT'' AND salary > 70000';

## Index Security and Permissions

### Index-Related Permissions

-- Permissions needed for index operations  
-- CREATE/DROP INDEX requires ALTER permission on table  
GRANT ALTER ON employees TO IndexManager;  
  
-- VIEW DEFINITION required to see index definitions  
GRANT VIEW DEFINITION ON employees TO Developer;  
  
-- Example: Create role for index management  
CREATE ROLE db\_indexmanager;  
  
-- Grant necessary permissions  
GRANT ALTER ON SCHEMA::dbo TO db\_indexmanager;  
GRANT VIEW DEFINITION ON SCHEMA::dbo TO db\_indexmanager;  
  
-- Add user to role  
ALTER ROLE db\_indexmanager ADD MEMBER [domain\indexadmin];

### Monitoring Index Security

-- Check permissions on indexes  
SELECT   
 p.principal\_id,  
 pr.name AS principal\_name,  
 p.permission\_name,  
 p.state\_desc,  
 o.name AS object\_name,  
 i.name AS index\_name  
FROM sys.database\_permissions p  
JOIN sys.objects o ON p.major\_id = o.object\_id  
JOIN sys.database\_principals pr ON p.grantee\_principal\_id = pr.principal\_id  
LEFT JOIN sys.indexes i ON o.object\_id = i.object\_id  
WHERE p.permission\_name IN ('ALTER', 'VIEW DEFINITION')  
AND o.type = 'U';

## Index Maintenance Automation

### Automated Index Maintenance Job

-- Create comprehensive index maintenance procedure  
CREATE PROCEDURE sp\_AutomatedIndexMaintenance  
 @database\_name VARCHAR(128) = NULL,  
 @table\_name VARCHAR(128) = NULL,  
 @fragmentation\_threshold\_reorganize FLOAT = 10.0,  
 @fragmentation\_threshold\_rebuild FLOAT = 30.0,  
 @min\_page\_count INT = 1000,  
 @max\_duration\_minutes INT = 240,  
 @online\_rebuild BIT = 1  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 IF @database\_name IS NULL SET @database\_name = DB\_NAME();  
   
 DECLARE @start\_time DATETIME = GETDATE();  
 DECLARE @sql NVARCHAR(MAX);  
 DECLARE @msg NVARCHAR(255);  
   
 -- Create temp table for maintenance tasks  
 CREATE TABLE #MaintenanceTasks (  
 id INT IDENTITY(1,1),  
 database\_name VARCHAR(128),  
 schema\_name VARCHAR(128),  
 table\_name VARCHAR(128),  
 index\_name VARCHAR(128),  
 fragmentation\_percent FLOAT,  
 page\_count BIGINT,  
 action\_type VARCHAR(20),  
 sql\_command NVARCHAR(MAX),  
 executed BIT DEFAULT 0,  
 execution\_time DATETIME NULL,  
 error\_message NVARCHAR(MAX) NULL  
 );  
   
 -- Populate maintenance tasks  
 INSERT INTO #MaintenanceTasks (database\_name, schema\_name, table\_name, index\_name,   
 fragmentation\_percent, page\_count, action\_type, sql\_command)  
 SELECT   
 @database\_name,  
 OBJECT\_SCHEMA\_NAME(ips.object\_id),  
 OBJECT\_NAME(ips.object\_id),  
 i.name,  
 ips.avg\_fragmentation\_in\_percent,  
 ips.page\_count,  
 CASE   
 WHEN ips.avg\_fragmentation\_in\_percent >= @fragmentation\_threshold\_rebuild THEN 'REBUILD'  
 WHEN ips.avg\_fragmentation\_in\_percent >= @fragmentation\_threshold\_reorganize THEN 'REORGANIZE'  
 END,  
 CASE   
 WHEN ips.avg\_fragmentation\_in\_percent >= @fragmentation\_threshold\_rebuild THEN  
 'ALTER INDEX ' + QUOTENAME(i.name) + ' ON ' +   
 QUOTENAME(OBJECT\_SCHEMA\_NAME(ips.object\_id)) + '.' + QUOTENAME(OBJECT\_NAME(ips.object\_id)) +   
 ' REBUILD' + CASE WHEN @online\_rebuild = 1 THEN ' WITH (ONLINE = ON)' ELSE '' END  
 WHEN ips.avg\_fragmentation\_in\_percent >= @fragmentation\_threshold\_reorganize THEN  
 'ALTER INDEX ' + QUOTENAME(i.name) + ' ON ' +   
 QUOTENAME(OBJECT\_SCHEMA\_NAME(ips.object\_id)) + '.' + QUOTENAME(OBJECT\_NAME(ips.object\_id)) +   
 ' REORGANIZE'  
 END  
 FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(@database\_name), OBJECT\_ID(@table\_name), NULL, NULL, 'SAMPLED') ips  
 JOIN sys.indexes i ON ips.object\_id = i.object\_id AND ips.index\_id = i.index\_id  
 WHERE i.name IS NOT NULL  
 AND ips.page\_count >= @min\_page\_count  
 AND ips.avg\_fragmentation\_in\_percent >= @fragmentation\_threshold\_reorganize  
 ORDER BY ips.avg\_fragmentation\_in\_percent DESC;  
   
 -- Execute maintenance tasks  
 DECLARE @task\_id INT, @task\_sql NVARCHAR(MAX), @task\_action VARCHAR(20);  
 DECLARE @task\_table VARCHAR(128), @task\_index VARCHAR(128);  
   
 DECLARE maintenance\_cursor CURSOR FOR  
 SELECT id, sql\_command, action\_type, table\_name, index\_name  
 FROM #MaintenanceTasks  
 WHERE executed = 0  
 ORDER BY fragmentation\_percent DESC;  
   
 OPEN maintenance\_cursor;  
 FETCH NEXT FROM maintenance\_cursor INTO @task\_id, @task\_sql, @task\_action, @task\_table, @task\_index;  
   
 WHILE @@FETCH\_STATUS = 0 AND DATEDIFF(MINUTE, @start\_time, GETDATE()) < @max\_duration\_minutes  
 BEGIN  
 BEGIN TRY  
 SET @msg = 'Executing ' + @task\_action + ' on ' + @task\_table + '.' + @task\_index;  
 RAISERROR(@msg, 10, 1) WITH NOWAIT;  
   
 EXEC sp\_executesql @task\_sql;  
   
 UPDATE #MaintenanceTasks   
 SET executed = 1, execution\_time = GETDATE()  
 WHERE id = @task\_id;  
   
 END TRY  
 BEGIN CATCH  
 UPDATE #MaintenanceTasks   
 SET error\_message = ERROR\_MESSAGE()  
 WHERE id = @task\_id;  
   
 SET @msg = 'Error executing ' + @task\_action + ' on ' + @task\_table + '.' + @task\_index + ': ' + ERROR\_MESSAGE();  
 RAISERROR(@msg, 16, 1) WITH NOWAIT;  
 END CATCH  
   
 FETCH NEXT FROM maintenance\_cursor INTO @task\_id, @task\_sql, @task\_action, @task\_table, @task\_index;  
 END  
   
 CLOSE maintenance\_cursor;  
 DEALLOCATE maintenance\_cursor;  
   
 -- Summary report  
 SELECT   
 'Maintenance Summary' AS report\_section,  
 COUNT(\*) AS total\_tasks,  
 SUM(CASE WHEN executed = 1 THEN 1 ELSE 0 END) AS completed\_tasks,  
 SUM(CASE WHEN error\_message IS NOT NULL THEN 1 ELSE 0 END) AS failed\_tasks,  
 DATEDIFF(MINUTE, @start\_time, GETDATE()) AS duration\_minutes  
 FROM #MaintenanceTasks  
   
 UNION ALL  
   
 SELECT   
 action\_type + ' Tasks',  
 COUNT(\*),  
 SUM(CASE WHEN executed = 1 THEN 1 ELSE 0 END),  
 SUM(CASE WHEN error\_message IS NOT NULL THEN 1 ELSE 0 END),  
 AVG(CASE WHEN executed = 1 THEN DATEDIFF(SECOND, @start\_time, execution\_time) ELSE NULL END)  
 FROM #MaintenanceTasks  
 GROUP BY action\_type;  
   
 -- Detailed results  
 SELECT \* FROM #MaintenanceTasks ORDER BY fragmentation\_percent DESC;  
   
 DROP TABLE #MaintenanceTasks;  
END  
  
-- Execute automated maintenance  
EXEC sp\_AutomatedIndexMaintenance   
 @fragmentation\_threshold\_reorganize = 10.0,  
 @fragmentation\_threshold\_rebuild = 30.0,  
 @online\_rebuild = 1;

## Summary

Indexes are crucial for SQL Server performance optimization:

### Key Takeaways:

* **Clustered indexes** determine physical data storage order
* **Non-clustered indexes** provide fast data lookup paths
* **Composite indexes** support complex query patterns
* **Covering indexes** eliminate key lookups
* **Filtered indexes** optimize storage for specific data subsets
* **Columnstore indexes** excel at analytical workloads

### Best Practices Summary:

1. **Design indexes based on query patterns**, not just table structure
2. **Monitor index usage** and remove unused indexes
3. **Maintain indexes regularly** to prevent fragmentation
4. **Use covering indexes** to avoid key lookups
5. **Consider filtered indexes** for selective data
6. **Balance read performance** against write overhead
7. **Test index changes** in non-production environments first

### Performance Impact:

* **Proper indexing** can improve query performance by 10-1000x
* **Poor indexing** can slow down DML operations significantly
* **Regular maintenance** prevents performance degradation over time

Master these indexing concepts to build high-performance SQL Server applications!