Understanding Indexing in MSSQL

What is Indexing in MSSQL?

Indexing in MSSQL (Microsoft SQL Server) is a database optimization technique used to improve the speed of data retrieval. It creates a data structure (similar to the index in a book) that helps the database engine locate rows more efficiently.

Types of Indexes in MSSQL

1. **Clustered Index**:
    - A clustered index determines the physical order of data in the table.
    - There can only be one clustered index per table since the data can only be sorted in one way.
2. **Example**:

```
CREATE CLUSTERED INDEX IX_Products_ProductID ON Products(ProductID);
```

Non-Clustered Index:

- A non-clustered index creates a separate structure from the table data, with pointers to the actual rows.
- Multiple non-clustered indexes can be created on a single table.

Example:

```
CREATE NONCLUSTERED INDEX IX_Products_ProductName ON
Products(ProductName);
```

Full-Text Index:

- Used for advanced text-based searches, such as searching for keywords in large text columns.

Example:

```
CREATE FULLTEXT INDEX ON Products(ProductName)

KEY INDEX IX_Products_ProductID;
```

Benefits of Indexing

1. **Faster Query Performance**:
    - Indexes speed up queries by reducing the amount of data scanned.
2. **Efficient Sorting**:
    - Queries with `ORDER BY` clauses benefit from indexes.
3. **Improved Filtering**:
    - `WHERE` clauses can quickly locate rows matching criteria.

What Happens When You Create an Index?

Clustered Index:

- It **reorganizes the actual table data** to match the index order. The table itself is stored in the sorted order of the indexed column(s).
- Think of it as sorting the rows of the table by the indexed column (like sorting a book alphabetically by title).

Non-Clustered Index:

- It creates a **separate structure** (like a lookup table) that stores the indexed column(s) and pointers to the actual rows in the table.
- The table data remains unchanged, but the index provides a faster way to locate rows based on the indexed column.

2. Does It Create a JSON File or Table?

- **No JSON File**:
  - MSSQL does not use JSON files for indexing. The index is stored in the database in its own internal structure.
- **No New Table**:
  - A new table is not created. The index is a special data structure linked to the table it belongs to.

Instead, the index is stored as **metadata** and **data pages** in the database files (`.mdf` and `.ndf` files).

3. What Does the Index Look Like Internally?

Indexes are typically implemented as a **B-tree** structure in MSSQL:

- **Clustered Index**:
  - The table data itself is stored in the B-tree structure.
  - Each leaf node contains the actual table data.
- **Non-Clustered Index**:
  - The B-tree contains pointers (references) to the actual data in the table.

Example of a Non-Clustered Index:

If you create an index on `ProductName`, the index might look like this internally:

Index:

- Apple → Pointer to Row 3

- Banana → Pointer to Row 1

- Cherry → Pointer to Row 2

The actual table remains unchanged:

Table:

Row 1: Banana

Row 2: Cherry

Row 3: Apple

4. Where Is the Index Stored?

- Indexes are stored in the same **database files** (`.mdf` or `.ndf`).
- When you query or rebuild an index, SQL Server uses these files to manage the index.

5. What Does It Mean for Storage?

- Indexes take up **additional storage space** within the database.
    - Clustered Index: Reorganizes the table, so no additional space is required for the index itself (but sorting may increase fragmentation).
    - Non-Clustered Index: Requires extra space for the index structure since it is stored separately from the table.

When you create an index in MSSQL:

- It creates an **internal data structure** (like a B-tree).
- No new table or JSON file is created.
- The index is stored within the database files and works alongside the table to improve query performance.

Challenges of Indexing

1. **Performance Overhead**:
    - Indexes slow down `INSERT`, `UPDATE`, and `DELETE` operations because the index also needs to be updated.
2. **Storage Costs**:
    - Indexes consume additional disk space.
3. **Maintenance**:
    - Indexes can become fragmented and require periodic rebuilding.

Part 2: Introduction to Elasticsearch

What is Elasticsearch?

Elasticsearch is a distributed, open-source search and analytics engine designed for scalability and real-time data access. Unlike MSSQL, which is optimized for structured data, Elasticsearch excels in:

- **Full-text search**: Finding documents based on keywords and relevance.
- **Unstructured or semi-structured data**: JSON-based storage.
- **Distributed Architecture**: Handles petabytes of data across clusters.

Why Use Elasticsearch?

1. **Full-Text Search**:
   - Advanced capabilities like fuzzy search, autocomplete, and relevance scoring.
2. **Real-Time Analytics**:
   - Perform aggregations and queries on massive datasets.
3. **Scalability**:
   - Elastic scales horizontally by adding nodes to the cluster.
4. **JSON Document Storage**:
   - Stores data in a schema-free JSON format.

How Elasticsearch Works

1. **Index**:
   - Similar to a database in SQL, an Elasticsearch index is a collection of documents.
2. **Document**:
   - Each record (stored as JSON) in an Elasticsearch index is a document.
3. **Shard**:
   - Indexes are divided into smaller units called shards for distribution.
4. **REST API**:
   - Elasticsearch provides RESTful APIs for data ingestion, search, and analytics.

Part 3: Using MSSQL and Elasticsearch in Docker

Step 1: Running MSSQL in Docker

1. Pull and run the MSSQL Docker image:

```
docker pull mcr.microsoft.com/mssql/server:2019-latest
```

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=YourStrongPassword123' -p 1433:1433 --name mssql-container -d mcr.microsoft.com/mssql/server:2019-latest
```

Connect to MSSQL and create the `Products` table:

```
CREATE DATABASE ProductDB;

USE ProductDB;


CREATE TABLE Products (
        ProductID INT PRIMARY KEY,
        ProductName NVARCHAR(100),
        Price DECIMAL(10, 2),
        Stock INT
);


INSERT INTO Products (ProductID, ProductName, Price, Stock) VALUES
(1, 'Laptop', 1200.00, 50),
(2, 'Mouse', 25.00, 200),
(3, 'Keyboard', 45.00, 100);
```

Add an index to the `ProductName` column:

CREATE NONCLUSTERED INDEX IX_Products_ProductName ON Products(ProductName);

Step 2: Running Elasticsearch in Docker

1. Pull and run the Elasticsearch Docker image:

docker pull docker.elastic.co/elasticsearch/elasticsearch:8.10.0

docker run -d --name elasticsearch -p 9200:9200 -e "discovery.type=single-node" docker.elastic.co/elasticsearch/elasticsearch:8.10.0

Verify Elasticsearch is running:

- Open a browser and go to `http://localhost:9200`.

Step 3: Integrating MSSQL and Elasticsearch in .NET Web API

1. Create a .NET Web API Project

1. Create a new Web API project:

```
dotnet new webapi -n ProductSearchAPI

cd ProductSearchAPI
```

Install required NuGet packages:

```
dotnet add package Microsoft.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package NEST
```

2. Configure MSSQL

1. Add the MSSQL connection string to `appsettings.json`:

```
{
    "ConnectionStrings": {
    "DefaultConnection": "Server=localhost,1433;Database=ProductDB;User Id=SA;Password=YourStrongPassword123;"
    },
    "Elasticsearch": {
    "Uri": "http://localhost:9200"
    }
}
```

Create the `Product` model and database context:

```
public class Product
{
	public int ProductID { get; set; }
	public string ProductName { get; set; }
	public decimal Price { get; set; }
	public int Stock { get; set; }
}
public class ProductContext : DbContext
{
	public ProductContext(DbContextOptions<ProductContext> options) : base(options) { }
	public DbSet<Product> Products { get; set; }
}
```

Register the context in `Program.cs`:

```
builder.Services.AddDbContext<ProductContext>(options =>
        options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Configure Elasticsearch

1. Create an `ElasticsearchService`:

```
using Nest;
public class ElasticsearchService
{       private readonly ElasticClient _client;
        public ElasticsearchService(IConfiguration configuration)
        {
        var settings = new ConnectionSettings(new Uri(configuration["Elasticsearch:Uri"]))
        .DefaultIndex("products");
        _client = new ElasticClient(settings);
        }
        public async Task IndexProductAsync(Product product)
        {
        await _client.IndexDocumentAsync(product);
        }
```

```
        public async Task<ISearchResponse<Product>> SearchProductsAsync(string query)
        {
        return await _client.SearchAsync<Product>(s => s
        .Query(q => q
                .Match(m => m
                .Field(f => f.ProductName)
                .Query(query)
                )
        )
        );
        }
}
```

Register the Elasticsearch service in `Program.cs`:

```
builder.Services.AddSingleton<ElasticsearchService>();
```

4. Create API Endpoints

1. Add a `SearchController`:

```
[ApiController]
[Route("api/[controller]")]
public class SearchController : ControllerBase
{
        private readonly ElasticsearchService _elasticService;

        public SearchController(ElasticsearchService elasticService)
        {
        _elasticService = elasticService;
        }
```

```
        [HttpPost("index")]
        public async Task<IActionResult> IndexProduct([FromBody] Product product)
        {
        await _elasticService.IndexProductAsync(product);
        return Ok("Product indexed successfully.");
        }
        [HttpGet("search")]
        public async Task<IActionResult> Search([FromQuery] string query)
        {
        var result = await _elasticService.SearchProductsAsync(query);
        return Ok(result.Documents);
        }
}
```

Testing

1. **Index a product into Elasticsearch**:

```
POST /api/search/index

Content-Type: application/json

{

    "productID": 4,

    "productName": "Monitor",

    "price": 250,

    "stock": 30

}
```

Search for products:

GET /api/search/search?query=Monitor