

Data Management Using Microsoft SQL Server

Session: 11

Indexes

For Aptech Centre Use Only

Objectives

- Define and explain indexes
- Describe the performance of indexes
- Explain clustered indexes
- Explain nonclustered indexes
- Explain partitioning of data
- Explain the steps to display query performance data using indexes

For Aptech Centre Use Only

Introduction

- SQL Server 2012 makes use of indexes to find data when a query is processed.
- When SQL Server has not defined any index for searching, then the SQL engine needs to visit every row in a table.
- As a table grows up to thousands and millions of rows and beyond, scans become slower and more expensive.
- In such cases, indexes are strongly recommended.
- Creating or removing indexes from a database schema will not affect an application's code.
- Indexes operate in the backend with support of the database engine.
- Moreover, creating an appropriate index can significantly increase the performance of an application.

Overview of Data Storage 1-2

A book contains pages, which contain paragraphs made up of sentences. Similarly, SQL Server 2012 stores data in storage units known as data pages.

These pages contain data in the form of rows. In SQL Server 2012, the size of each data page is 8 Kilo Bytes (KB).

Thus, SQL Server databases have 128 data pages per Mega Byte (MB) of storage space. A page begins with a 96-byte header, which stores system information about the page.

➤ This information includes the following:

Page number

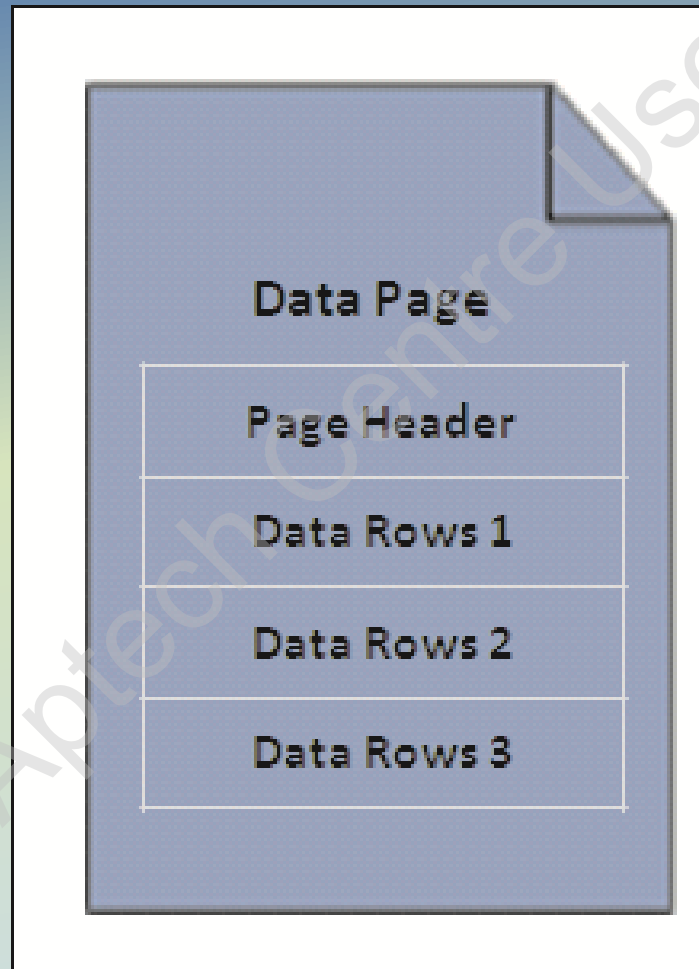
Page type

Amount of free space on the page

Allocation unit ID of the object to which the page is allocated

Overview of Data Storage 2-2

- Following figure shows data storage structure of a data page:



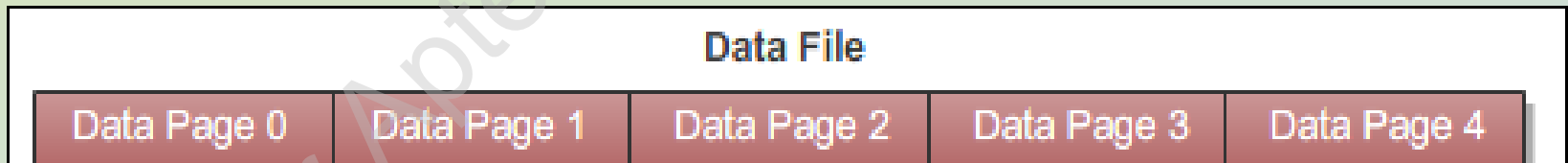
Data Files 1-2

All input and output operations in the database are performed at the page level. This means that the database engine reads or writes data pages.

A set of eight contiguous data pages is referred to as an extent. SQL Server 2012 stores data pages in files known as data files.

The space allotted to a data file is divided into sequentially numbered data pages.

- The numbering starts from zero as shown in the following figure:



Data Files 2-2

- There are three types of data files in SQL Server 2012. These are as follows:

Primary Data Files

- A primary data file is automatically created at the time of creation of the database.
- This file has references to all other files in the database.
- The recommended file extension for primary data files is `.mdf`.

Secondary Data Files

- Secondary data files are optional in a database and can be created to segregate database objects such as tables, views, and procedures.
- The recommended file extension for secondary data files is `.ndf`.

Log Files

- Log files contain information about modifications carried out in the database.
- This information is useful in recovery of data in contingencies such as sudden power failure or the need to shift the database to a different server.
- There is at least one log file for each database.
- The recommended file extension for log files is `.ldf`.

Requirement for Indexes

- To facilitate quick retrieval of data from a database, SQL Server 2012 provides the indexing feature.
- An index in SQL Server 2012 database contains information that allows you to find specific data without scanning through the entire table as shown in the following figure:

| Index | | | |
|-----------|----|---------------|----|
| A | | | |
| Adapter | 1 | Border | 19 |
| Aggregate | 10 | Bullet | 58 |
| Analysis | 13 | | |
| Average | 23 | | |
| B | | C | |
| Board | 17 | Consistency | 20 |
| Brilliant | 18 | Connect | 22 |
| | | Communication | 24 |
| | | Character | 30 |

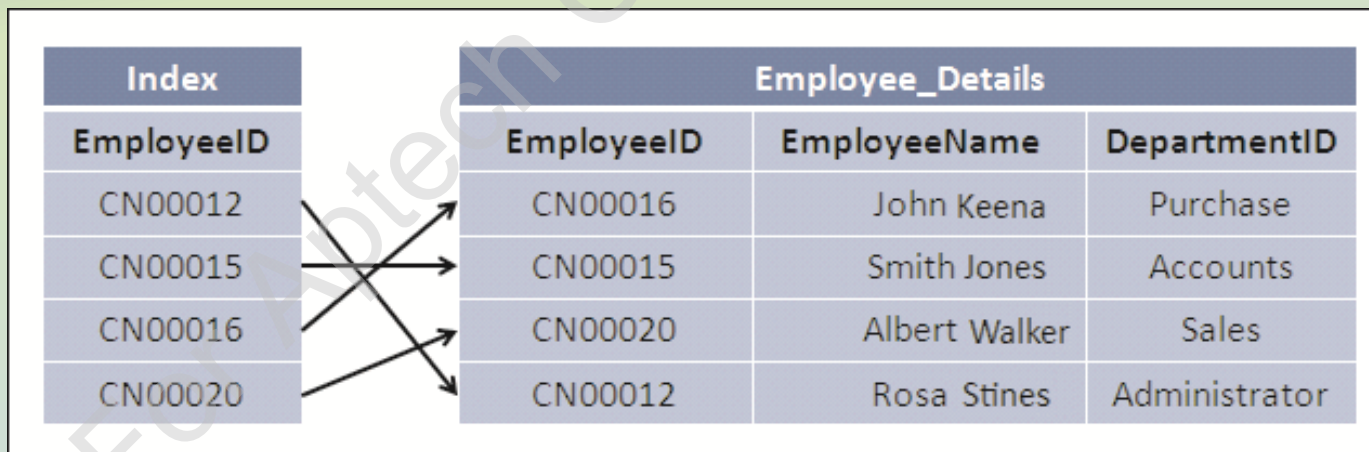
Indexes 1-3

In a table, records are stored in the order in which they are entered. Their storage in the database is unsorted.

When data is to be retrieved from such tables, the entire table needs to be scanned.

This slows down the query retrieval process. To speed up query retrieval, indexes need to be created.

When an index is created on a table, the index creates an order for the data rows or records in the table as shown in the following figure:



- This assists in faster location and retrieval of data during searches.

Indexes 2-3

- Indexes are automatically created when PRIMARY KEY and UNIQUE constraints are defined on a table.
- Indexes reduce disk I/O operations and consume fewer system resources.
- The CREATE INDEX statement is used to create an index.
- The syntax for creating an index is as follows:

Syntax:

```
CREATE INDEX <index_name> ON <table_name> (<column_name>)
```

where,

index_name: specifies the name of the index.

table_name: specifies the name of the table.

column_name: specifies the name of the column.

- Following code snippet creates an index, **IX_Country** on the **Country** column in the **Customer_Details** table:

```
USE CUST_DB
CREATE INDEX IX_Country ON Customer_Details(Country);
GO
```

Indexes 3-3

- Following figure shows the indexed table of **Customer_Details**:

| Customer_Details | | | | Index |
|------------------|-------|---------------|---------|------------|
| CustID | AccNo | AccName | Country | IX_Country |
| 01 | CN001 | John Keena | Spain | Germany |
| 02 | CN020 | Smith Jones | Russia | London |
| 03 | CN011 | Albert Walker | Germany | Russia |
| 04 | CN021 | Rosa Stines | London | Spain |

- Indexes point to the location of a row on a data page instead of searching through the table.
- Consider the following facts and guidelines about indexes:

Indexes increase the speed of queries that join tables or perform sorting operations.

Indexes implement the uniqueness of rows if defined when you create an index.

Indexes are created and maintained in ascending or descending order.

Scenario

- In a telephone directory, where a large amount of data is stored and is frequently accessed, the storage of data is done in an alphabetical order.
- If such data were unsorted, it would be nearly impossible to search for a specific telephone number.
- Similarly, in a database table having a large number of records that are frequently accessed, the data is to be sorted for fast retrieval.
- When an index is created on the table, the index either physically or logically sorts the records.
- Thus, searching for a specific record becomes faster and there is less strain on system resources.

Accessing Data Group-wise

- Indexes are useful when data needs to be accessed group-wise.
- For example, you want to make modifications to the conveyance allowance for all employees based on the department they work in.
- Here, you wish to make the changes for all employees in one department before moving on to employees in another department.
- In this case, an index can be created as shown in the following figure on the Department column before accessing the records:

| Department Name | Employee Name |
|-----------------|-------------------|
| Marketing | Jenny Woods |
| Marketing | Merry Thomas |
| Marketing | John Updeeke |
| Marketing | Robert Williamson |
| Sales | Smith Gordon |
| Sales | Albert Wang |

- This index will create logical chunks of data rows based on the department.
- This again will limit the amount of data actually scanned during query retrieval.
- Hence, retrieval will be faster and there will be less strain on system resources.

Index Architecture

- In SQL Server 2012, data in the database can be stored either in a sorted manner or at random.
- If data is stored in a sorted manner, the data is said to be present in a clustered structure.
- If it is stored at random, it is said to be present in a heap structure.
- Following figure shows an example demonstrating index architecture:

| Employee_Details | | |
|------------------|--------------|--------|
| EmpID | EmpName | DeptID |
| CN00020 | Rosa Stevens | BN0001 |
| CN00018 | John Updeeke | BN0020 |
| CN00019 | Smith Gordon | BN0021 |
| CN00012 | Robert Tyson | BN0011 |

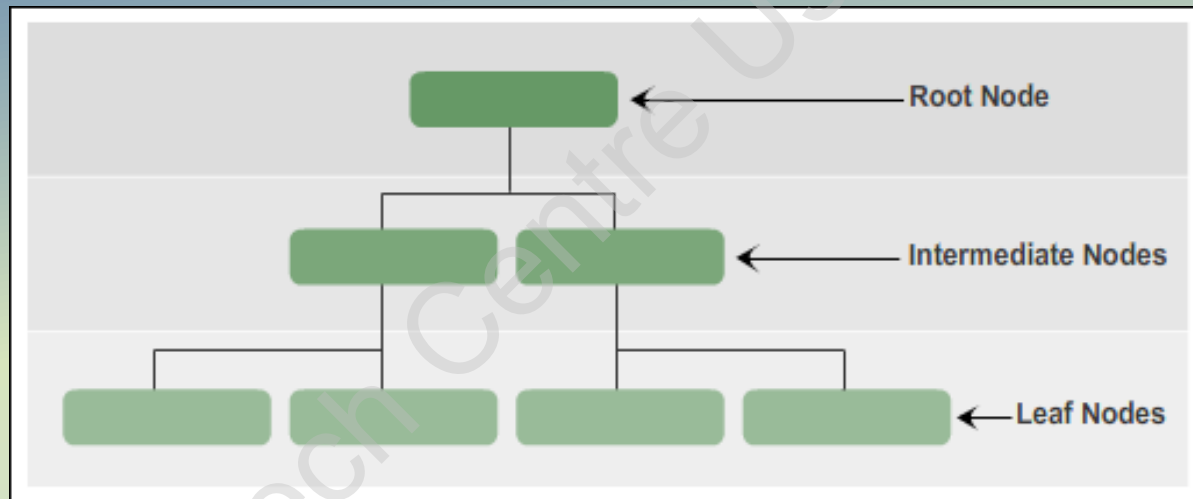
Heap Structure

| Employee_Details | | |
|------------------|--------------|--------|
| EmpID | EmpName | DeptID |
| CN00012 | Robert Tyson | BN0011 |
| CN00018 | John Updeeke | BN0020 |
| CN00019 | Smith Gordon | BN0021 |
| CN00020 | Rosa Stevens | BN0001 |

Clustered Structure

B-Tree

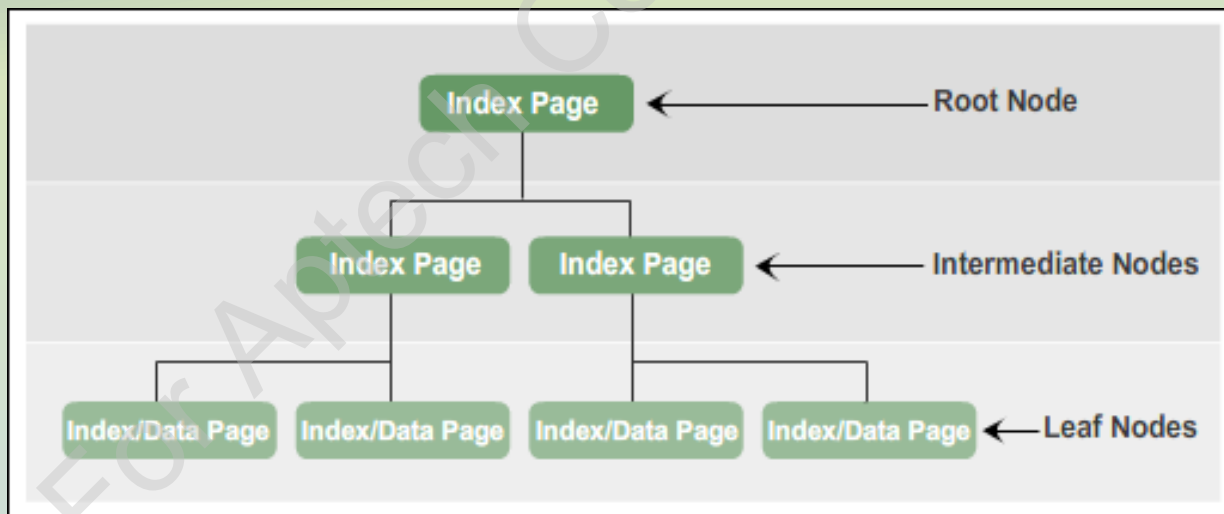
- In SQL Server, all indexes are structured in the form of B-Trees.
- A B-Tree structure can be visualized as an inverted tree with the root right at the top, splitting into branches and then, into leaves right at the bottom as shown in the following figure:



- In a B-Tree structure, there is a single root node at the top.
- This node then branches out into the next level, known as the first intermediate level.
- The nodes at the first intermediate level can branch out further.
- This branching can continue into multiple intermediate levels and then, finally the leaf level.
- The nodes at the leaf level are known as the leaf nodes.

Index B-Tree Structure 1-2

- In the B-Tree structure of an index, the root node consists of an index page.
- This index page contains pointers that point to the index pages present in the first intermediate level.
- These index pages in turn point to the index pages present in the next intermediate level.
- There can be multiple intermediate levels in an index B-Tree.
- The leaf nodes of the index B-Tree have either data pages containing data rows or index pages containing index rows that point to data rows as shown in the following figure:



Index B-Tree Structure 2-2

- Different types of nodes are as follows:

Root Node

- Contains an index page with pointers pointing to index pages at the first intermediate level.

Intermediate Nodes

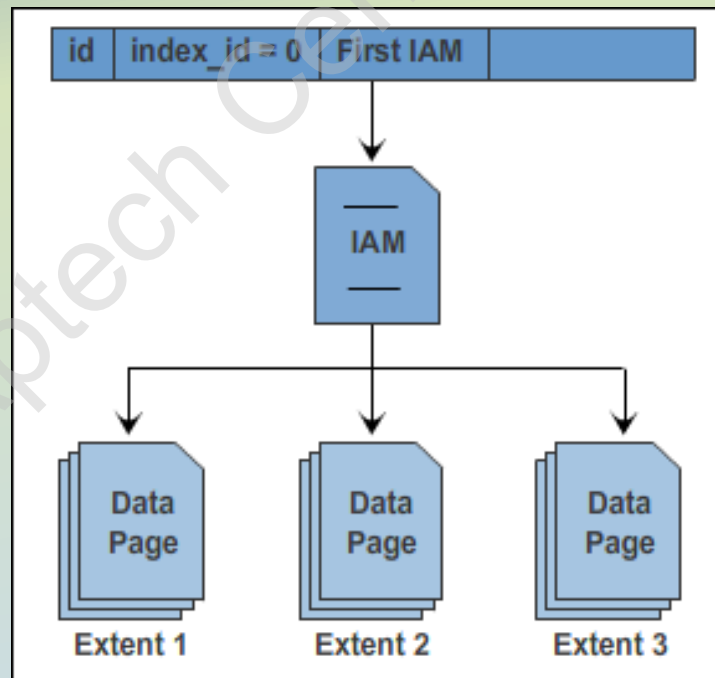
- Contain index pages with pointers pointing either to index pages at the next intermediate level or to index or data pages at the leaf level.

Leaf Nodes

- Contain either data pages or index pages that point to data pages.

Heap Structures

- In a heap structure, the data pages and records are not arranged in sorted order.
- The only connection between the data pages is the information recorded in the Index Allocation Map (IAM) pages.
- In SQL Server 2012, IAM pages are used to scan through a heap structure.
- IAM pages map extents that are used by an allocation unit in a part of a database file.
- A heap can be read by scanning the IAM pages to look for the extents that contain the pages for that heap as shown in the following figure:



Partitioning of Heap Structures

A table can be logically divided into smaller groups of rows.

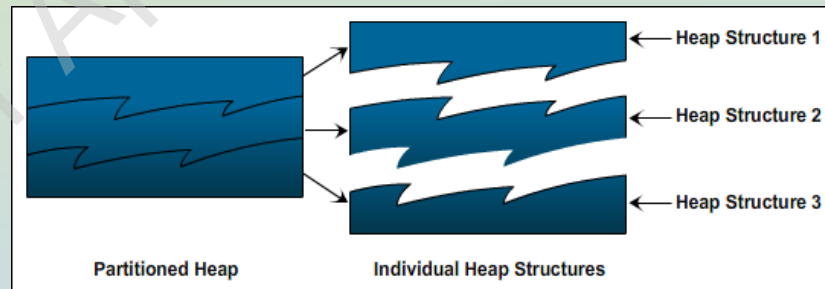
This division is referred to as partitioning.

Tables are partitioned in order to carry out maintenance operations more efficiently.

By default, a table has a single partition.

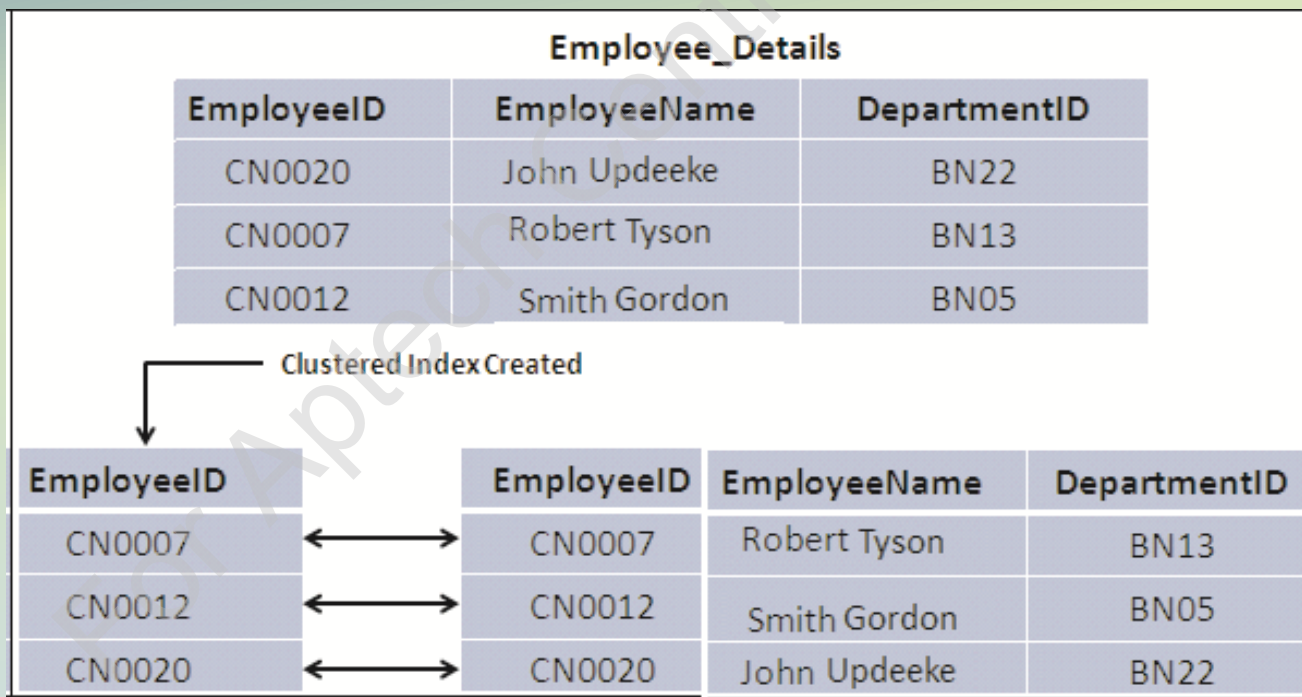
When partitions are created in a table with a heap structure, each partition will contain data in an individual heap structure.

For example, if a heap has three partitions, then there are three heap structures present, one in each partition as shown in the following figure:



Clustered Index Structures

- A clustered index causes records to be physically stored in a sorted or sequential order.
- A clustered index determines the actual order in which data is stored in the database. Hence, you can create only one clustered index in a table.
- Uniqueness of a value in a clustered index is maintained explicitly using the `UNIQUE` keyword or implicitly using an internal unique identifier as shown in the following figure:



Creating Clustered Index

- A clustered index causes records to be physically stored in a sorted or sequential order.
- You can create only one clustered index in a table.
- Clustered index is created using the `CREATE INDEX` statement with the `CLUSTERED` keyword.
- The syntax used to create a clustered index on a specified table is as follows:

Syntax:

```
CREATE CLUSTERED INDEX index_name ON <table_name> (column_name)
```

where,

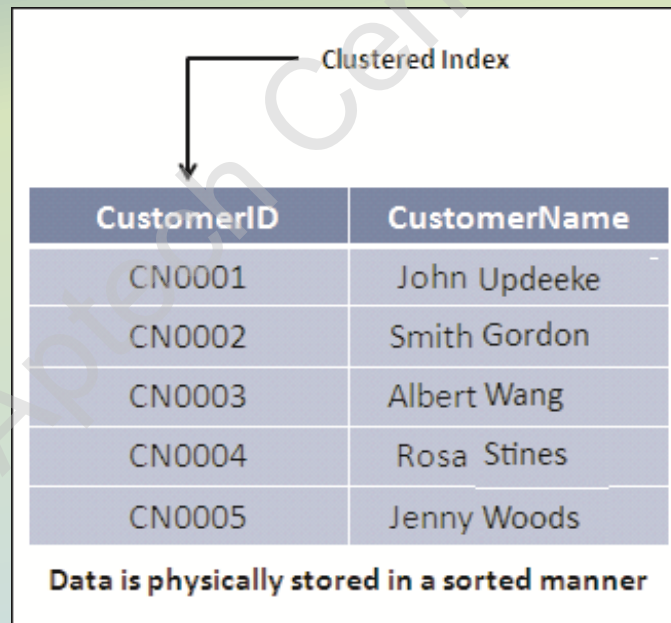
`CLUSTERED`: Specifies that a clustered index is created.

- Following code snippet creates a clustered index, **IX_CustID** on the **CustID** column in **Customer_Details** table:

```
USE CUST_DB
CREATE CLUSTERED INDEX IX_CustID ON Customer_Details(CustID)
GO
```

Accessing Data with a Clustered Index 1-2

- A clustered index can be created on a table using a column without duplicate values.
- This index reorganizes the records in the sequential order of the values in the index column.
- Clustered indexes are used to locate a single row or a range of rows.
- Starting from the first page of the index, the search value is checked against each key value on the page.
- When the matching key value is found, the database engine moves to the page indicated by that value as shown in the following figure:



| CustomerID | CustomerName |
|------------|--------------|
| CN0001 | John Updeeke |
| CN0002 | Smith Gordon |
| CN0003 | Albert Wang |
| CN0004 | Rosa Stines |
| CN0005 | Jenny Woods |

Data is physically stored in a sorted manner

- The desired row or range of rows is then accessed.

Accessing Data with a Clustered Index 2-2

- A clustered index is automatically created on a table when a primary key is defined on the table.
- In a table without a primary key column, a clustered index should ideally be defined on:

Key columns that are searched on extensively.

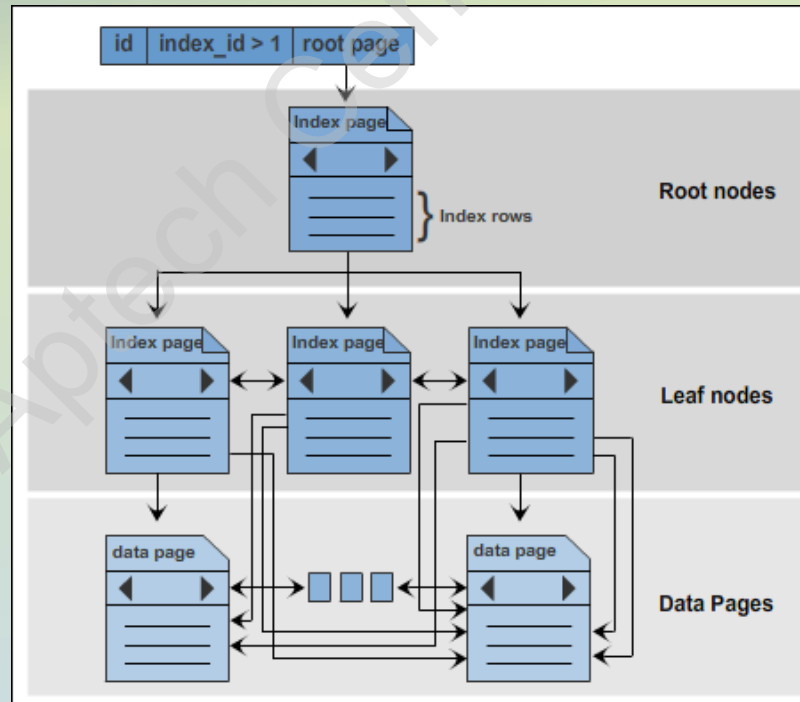
Columns used in queries that return large resultsets.

Columns having unique data.

Columns used in table join.

Nonclustered Index Structures 1-3

- A nonclustered index is defined on a table that has data in either a clustered structure or a heap.
- Nonclustered index will be the default type if an index is not defined on a table.
- Each index row in the nonclustered index contains a nonclustered key value and a row locator.
- This row locator points to the data row corresponding to the key value in the table.
- Following figure shows a nonclustered index structure:



Nonclustered Index Structures 2-3

- Nonclustered indexes have a similar B-Tree structure as clustered indexes but with the following differences:

The data rows of the table are not physically stored in the order defined by their nonclustered keys.

In a nonclustered index structure, the leaf level contains index rows.

- Nonclustered indexes are useful when you require multiple ways to search data.
- Some facts and guidelines to be considered before creating a nonclustered index are as follows:

When a clustered index is re-created or the `DROP_EXISTING` option is used, SQL Server rebuilds the existing nonclustered indexes.

A table can have up to 999 nonclustered indexes.

Create clustered index before creating a nonclustered index.

Nonclustered Index Structures 3-3

- The syntax used for creating a nonclustered index is as follows:

Syntax:

```
CREATE NONCLUSTERED INDEX <index_name> ON <table_name> (column_name)
```

where,

NONCLUSTERED: specifies that a nonclustered index is created.

- Following code snippet creates a nonclustered index **IX_State** on the **State** column in **Customer_Details** table:

```
USE CUST_DB  
CREATE NONCLUSTERED INDEX IX_State ON Customer_Details(State)  
GO
```

Column Store Index 1-5

Column Store Index is a new feature in SQL Server 2012.

It enhances performance of data warehouse queries extensively.

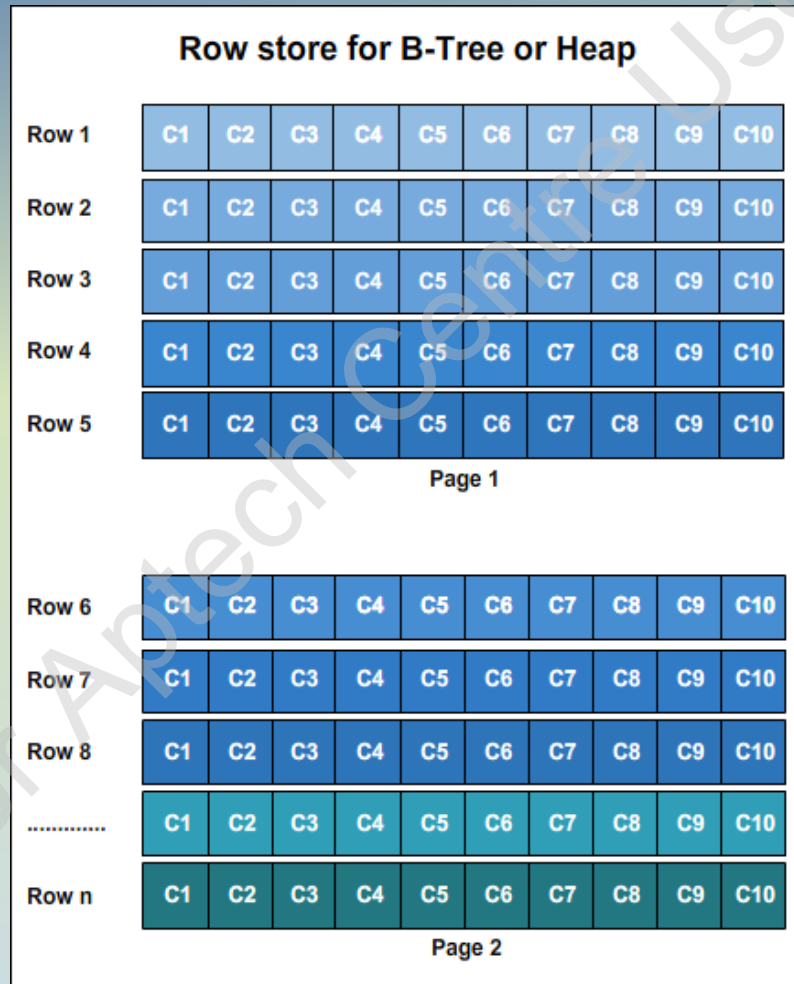
The regular indexes or heaps of older SQL Servers stored data in B-Tree structure row-wise, but the column store index in SQL Server 2012 stores data column-wise.

Since the data transfer rate is slow in database servers, so column store index uses compression aggressively to reduce the disk I/O needed to serve the query request.

The B-Tree and heap stores data row-wise, which means data from all the columns of a row are stored together contiguously on the same page.

Column Store Index 2-5

- For example, if there is a table with ten columns (C1 to C10), the data of all the ten columns from each row gets stored together contiguously on the same page as shown in the following figure:



Column Store Index 3-5

- When column store index is created, the data is stored column-wise, which means data of each individual column from each rows is stored together on same page.
- For example, the data of column C1 of all the rows gets stored together on one page and the data for column C2 of all the rows gets stored on another page and so on as shown in the following figure:

| Column Store Index | | | | | | | | | | |
|--------------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|---------|
| Row 1 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
| Row 2 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
| Row 3 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
| Row 4 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
| Row 5 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
| Row 6 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
| Row 7 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
| Row 8 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
| | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
| Row n | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 |
| | Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6 | Page 7 | Page 8 | Page 9 | Page 10 |

Column Store Index 4-5

- The syntax used to create a column store index is as follows:

Syntax:

```
CREATE [ NONCLUSTERED ] COLUMNSTORE INDEX index_name ON <object> (  
column [ ,...n ] )  
[ WITH ( <column_index_option> [ ,...n ] ) ]  
ON
```

- Assume that a table named **ResellerSalesPtnd** has been created in AdventureWorks2012 database.
- Following code snippet demonstrates how to create a column store index on this table:

```
USE CUST_DB  
CREATE NONCLUSTERED COLUMNSTORE INDEX [csindx_ResellerSalesPtnd]  
ON [ResellerSalesPtnd]  
(  
[ProductKey],  
[OrderDateKey],  
[DueDateKey],  
[ShipDateKey],  
[CustomerKey],  
[EmployeeKey],
```

Column Store Index 5-5

```
[PromotionKey],  
[CurrencyKey],  
[SalesTerritoryKey],  
[SalesOrderNumber],  
[SalesOrderLineNumber],  
[RevisionNumber],  
[OrderQuantity],  
[UnitPrice],  
[ExtendedAmount],  
[UnitPriceDiscountPct],  
[DiscountAmount],  
[ProductStandardCost],  
[TotalProductCost],  
[SalesAmount],  
[TaxAmt],  
[Freight],  
[CarrierTrackingNumber],  
[CustomerPONumber],  
[OrderDate],  
[DueDate],  
[ShipDate]  
);
```

Dropping an Index 1-3

While dropping all indexes on a table, you must first drop the nonclustered indexes first and then, the clustered index.

SQL Server 2012 can drop the clustered index and move the heap (unordered table) into another filegroup or a partition scheme using the `MOVE TO` option.

This option is not valid for nonclustered indexes.

The partition scheme or filegroup specified in the `MOVE TO` clause must already exist.

The table will be located in the same partition scheme or filegroup of the dropped clustered index.

Dropping an Index 2-3

- The syntax used to drop a clustered index is as follows:

Syntax:

```
DROP INDEX <index_name> ON <table_name>
[ WITH ( MOVE TO { <partition_scheme_name> ( <column_name> )
| <filegroup_name>
| 'default'
})
]
```

where,

index_name: specifies the name of the index.

partition_scheme_name: specifies the name of the partition scheme.

filegroup_name: specifies the name of the filegroup to store the partitions.

default: specifies the default location to store the resulting table.

Dropping an Index 3-3

- Following code snippet drops the index **IX_SuppID** created on the **SuppID** column of the **Supplier_Details** table:

```
DROP INDEX IX_SuppID ON Supplier_Details  
WITH (MOVE TO 'default')
```

- The data in the resulting **Supplier_Details** table is moved to the default location.
- Following code snippet drops the index **IX_SuppID** created on the **SuppID** column of the **Supplier_Details** table:

```
DROP INDEX IX_SuppID ON Supplier_Details  
WITH (MOVE TO FGCountry)
```

- The data in the resulting **Supplier_Details** table is moved to the **FGCountry** filegroup.



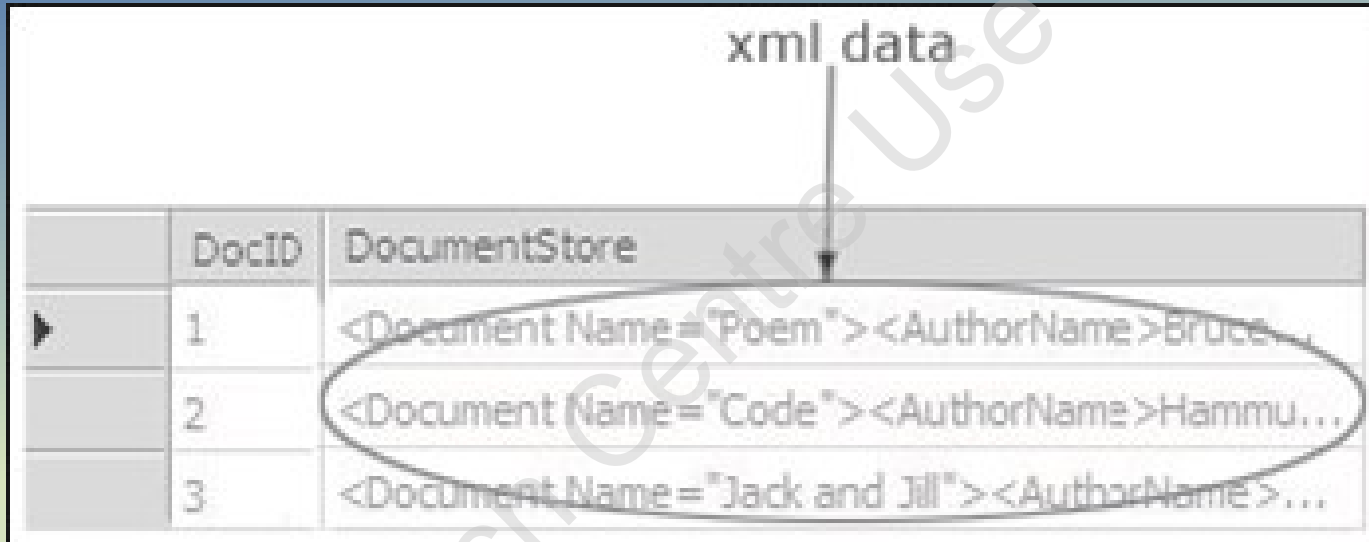
Difference between Clustered and Nonclustered Indexes

- Clustered and nonclustered indexes are different in terms of their architecture and their usefulness in query executions.
- Following table highlights the differences between clustered and nonclustered indexes:

| Clustered Indexes | Nonclustered Indexes |
|--|---|
| Used for queries that return large resultsets | Used for queries that do not return large resultsets |
| Only one clustered index can be created on a table | Multiple nonclustered indexes can be created on a table |
| The data is stored in a sorted manner on the clustered key | The data is not stored in a sorted manner on the nonclustered key |
| The leaf nodes of a clustered index contain the data pages | The leaf nodes of a nonclustered index contain index pages |

XML Indexes

- The xml data type is used to store XML documents and fragments as shown in the following figure:



| | DocID | DocumentStore |
|---|-------|--|
| ▶ | 1 | <Document Name = "Poem"><AuthorName>Bruce... |
| | 2 | <Document Name = "Code"><AuthorName>Hammu... |
| | 3 | <Document Name = "Jack and Jill"><AuthorName>... |

- An XML fragment is an XML instance that has a single top-level element missing.
- Due to the large size of XML columns, queries that search within these columns can be slow.
- You can speed up these queries by creating an XML index on each column.
- An XML index can be a clustered or a nonclustered index. Each table can have up to 249 XML indexes.

Types of XML Indexes 1-5

- XML indexes can be created on a table only if there is a clustered index based on the primary key of the table.
- This primary key cannot exceed 15 columns.
- The different types of XML indexes are as follows:

Primary XML Indexes

- The process of carrying out queries within an XML column can sometimes be slow.
- A primary XML index is created on each XML column to speed up these queries.
- It is a special index that shreds the XML data to store information.
- The syntax used to create a primary XML index is as follows:

Syntax:

```
CREATE PRIMARY XML INDEX index_name ON <table_name> (column_name)
```

Types of XML Indexes 2-5

- Following code snippet creates a primary XML index on the **CatalogDescription** column in the **Production.ProductModel** table:

```
USE AdventureWorks2012;  
CREATE PRIMARY XML INDEX PXML_ProductModel_CatalogDescription  
ON Production.ProductModel (CatalogDescription);  
GO
```

Secondary XML Indexes

- Secondary XML indexes are specialized XML indexes that help with specific XML queries.
- The features of secondary XML indexes are as follows:
 - Searching for values anywhere in the XML document.
 - Retrieving particular object properties from within an XML document.

Types of XML Indexes 3-5

- Secondary XML indexes can only be created on columns that already have a primary XML index.
- Following code snippet demonstrates how to create a secondary XML index on the **CatalogDescription** column in the **Production.ProductModel** table:

```
USE AdventureWorks2012;  
CREATE XML INDEX IXML_ProductModel_CatalogDescription_Path  
ON Production.ProductModel (CatalogDescription)  
USING XML INDEX PXML_ProductModel_CatalogDescription FOR PATH ;  
GO
```

Selective XML Indexes (SXI)

- This is a new type of XML index introduced by SQL Server 2012.
- The features of this new index is to improve querying performance over the data stored as XML in SQL Server, allows faster indexing of large XML data workloads, and improves scalability by reducing storage costs of the index.

Types of XML Indexes 4-5

- The syntax used to create selective XML index is as follows:

Syntax:

```
CREATE SELECTIVE XML INDEX index_name ON <table_name> (column_name)
```

- Following is an XML document in a table of approximately 500,000 rows:

```
<book>
  <created>2004-03-01</created>
  <authors>Various</authors>
  <subjects>
    <subject>English wit and humor -- Periodicals</subject>
    <subject>AP</subject>
  </subjects>
  <title>Punch, or the London Charivari, Volume 156, April 2,
  1919</title>
  <id>etext11617</id>
</book>
```


Types of XML Indexes 5-5

- Following code snippet demonstrates how to create a Selective XML index on the **BookDetails** column in the **BooksBilling** table:

```
USE CUST_DB
CREATE SELECTIVE XML INDEX SXI_index
ON BooksBilling(BookDetails)
FOR
(
  pathTitle = '/book/title/text()' AS XQUERY 'xs:string',
  pathAuthors = '/book/authors' AS XQUERY 'node()',
  pathId = '/book/id' AS SQL NVARCHAR(100)
)
GO
```

Modifying an XML Index

An XML index, primary or secondary, can be modified using the `ALTER INDEX` statement.

- The syntax used to modify an XML index is as follows:

Syntax:

```
ALTER INDEX <xml_index_name> ON <table_name> REBUILD
```

where,

`xml_index_name`: specifies the name of the XML index.

- Following code snippet rebuilds the primary XML index **PXML_DocumentStore** created on the **XMLDocument** table:

```
ALTER INDEX PXML_DocumentStore ON XMLDocument REBUILD
```

Removing an XML Index

- The syntax used to remove an XML index using the `DROP INDEX` statement is as follows:

Syntax:

```
DROP INDEX <xml_index_name> ON <table_name>
```

- Following code snippet removes the primary XML index **PXML_DocumentStore** created on the **XMLDocument** table:

```
DROP INDEX PXML_DocumentStore ON XMLDocument
```

Allocation Units 1-2

- A heap or a clustered index structure contains data pages in one or more allocation units.
- An allocation unit is a collection of pages and is used to manage data based on their page type.
- The types of allocation units that are used to manage data in tables and indexes are as follows:

IN_ROW_DATA

- It is used to manage data or index rows that contain all types of data except large object (LOB) data.

LOB_DATA

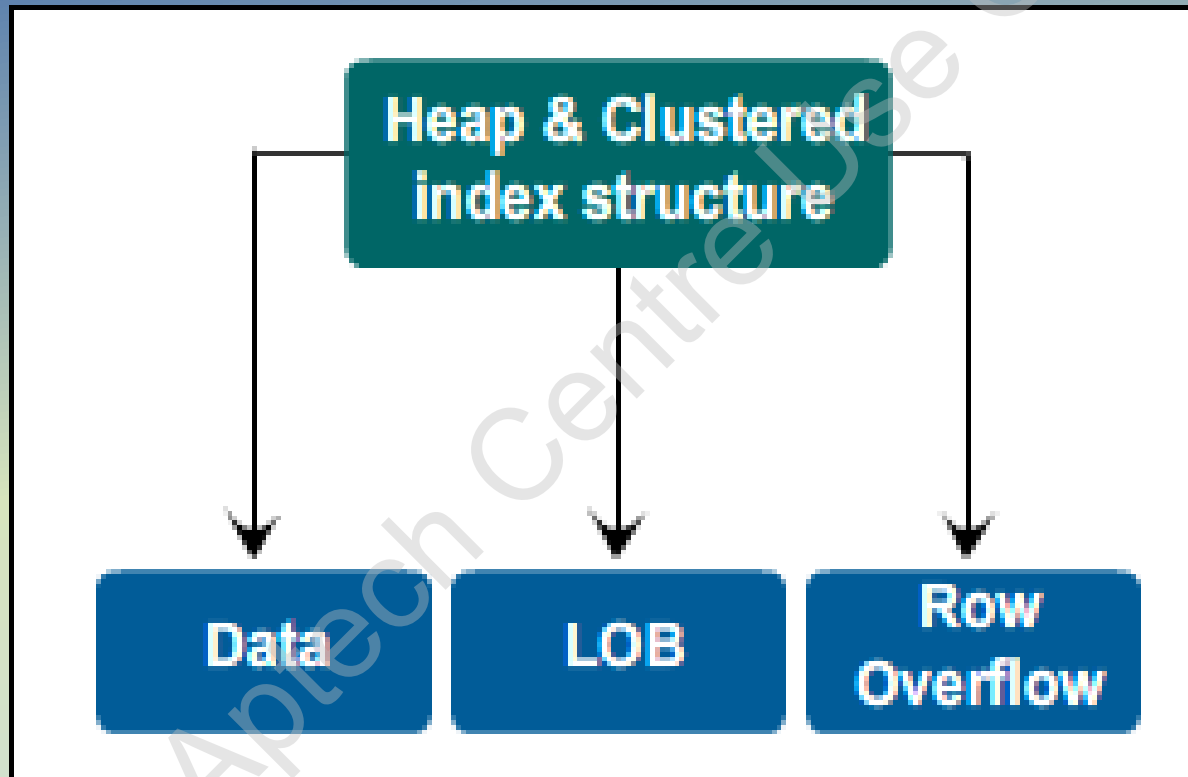
- It is used to manage large object data, which is stored in one or more of the following data types: `varbinary(max)`, `varchar(max)`, and `xml`.

ROW_OVERFLOW_DATA

- It is used to manage data of variable length, which is stored in `varchar`, `nvarchar`, `varbinary`, or `sql_variant` columns.

Allocation Units 2-2

- Following figure shows the allocation units:



Partitioning 1-2

Partitioning divides data into subsets.

This makes large tables or indexes more manageable.

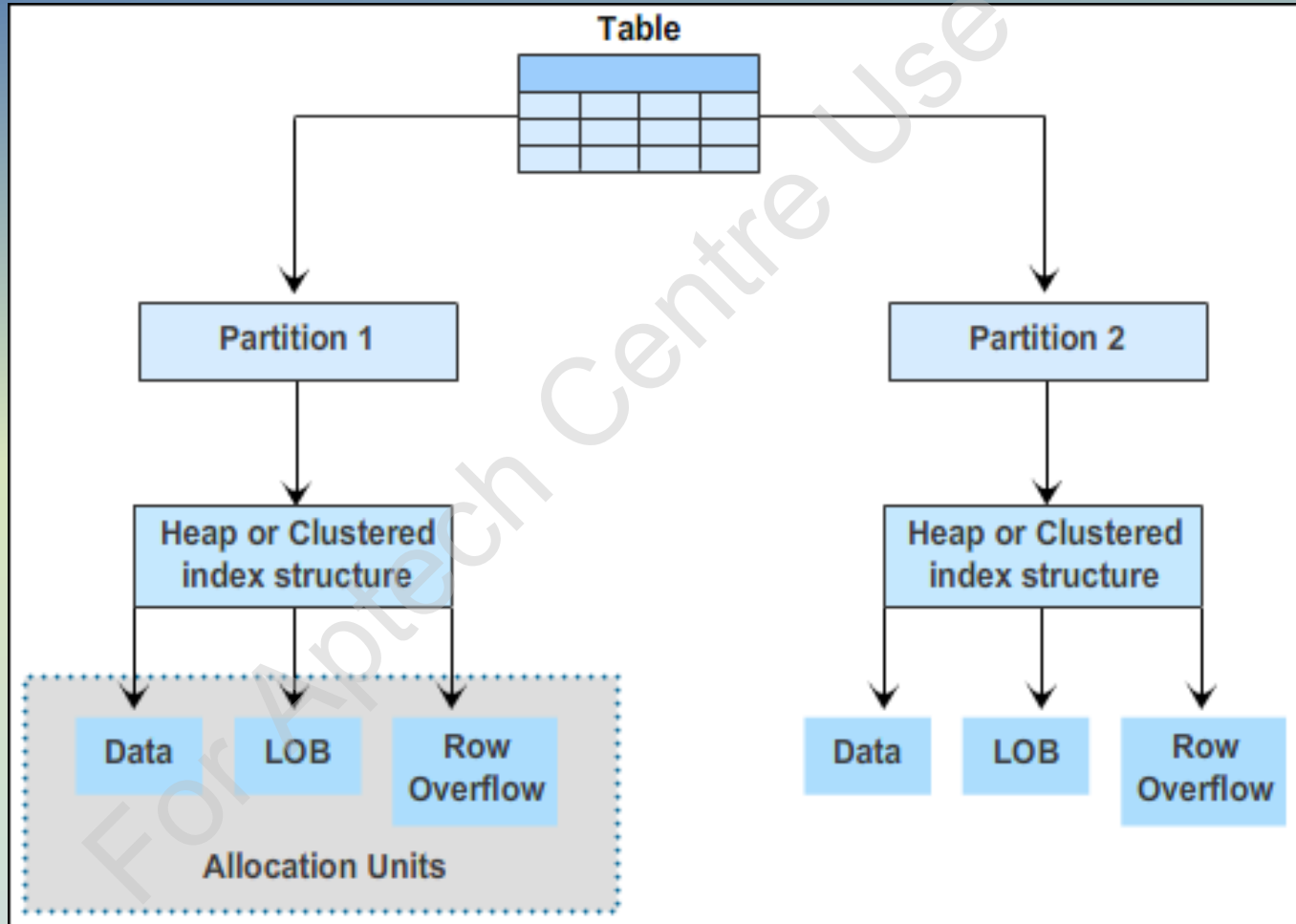
Partitioning enables you to access data quickly and efficiently.

Maintenance operations on subsets of data are performed more efficiently because they target only the required subset of data instead of the entire table.

By default, a table or an index has only one partition that contains all the data or index pages.

Partitioning 2-2

- When a table or index uses multiple partitions, the data is partitioned horizontally into groups of rows as shown in the following figure:



The sys.partitions View

- The `sys.partitions` view is a system view that contains the complete information about the different partitions of all the tables and indexes in the database.
- Following table shows the different columns of the `sys.partitions` view along with their data types and descriptions:

| Column Name | Data Type | Description |
|------------------|-----------|--|
| partition_id | bigint | Contains the id of the partition and is unique within a database. |
| object_id | int | Contains the id of the object to which the partition belongs. |
| index_id | int | Contains the id of the index to which the partition belongs. |
| partition_number | int | Contains the partition number within the index or heap. |
| hobt_id | bigint | Contains the id of the data heap or B-Tree that contains the rows for the partition. |
| rows | bigint | States the approximate number of rows in the partition. |

The `index_id` column 1-2

- The `index_id` column contains the id of the index to which the partition belongs.
- The `sys.partitions` catalog view returns a row for each partition in a table or index.
- The values of `index_id` column are unique within the table in which the partition is created.
- The data type of the `index_id` column is `int`.
- Following are the various values of the `index_id` column:

The `index_id` value for a heap is 0.

The `index_id` value for a clustered index is 1.

The `index_id` value for a nonclustered index is greater than 1.

The `index_id` value for large objects is greater than 250.

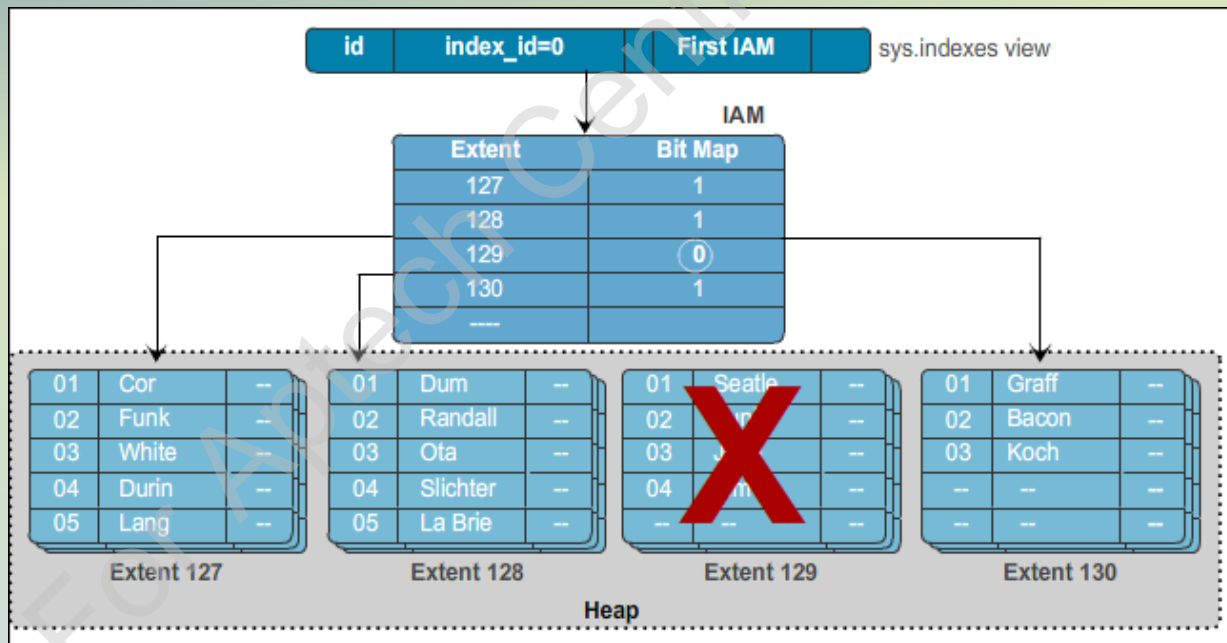
The index_id column 2-2

- Following figure shows the index_id column in the sys.partitions view:

| | partition_id | object_id | index_id | partition_number | hobt_id | rows |
|----|-----------------|-----------|----------|------------------|-----------------|------|
| 1 | 196608 | 3 | 1 | 1 | 196608 | 1779 |
| 2 | 327680 | 5 | 1 | 1 | 327680 | 332 |
| 3 | 458752 | 7 | 1 | 1 | 458752 | 379 |
| 4 | 524288 | 8 | 0 | 1 | 524288 | 2 |
| 5 | 281474977103872 | 6 | 1 | 1 | 281474977103872 | 0 |
| 6 | 281474977300480 | 9 | 1 | 1 | 281474977300480 | 0 |
| 7 | 281474977824768 | 17 | 1 | 1 | 281474977824768 | 0 |
| 8 | 281474977890304 | 18 | 1 | 1 | 281474977890304 | 0 |
| 9 | 281474977955840 | 19 | 1 | 1 | 281474977955840 | 0 |
| 10 | 281474978021376 | 20 | 1 | 1 | 281474978021376 | 2 |

Finding Rows 1-2

- SQL Server uses catalog views to find rows when an index is not created on a table.
- It uses the `sys.indexes` view to find the IAM page.
- This IAM page contains a list of all pages of a specific table through which SQL Server can read all data pages.
- When the `sys.indexes` view is used, the query optimizer checks all rows in a table and extracts only those rows that are referenced in the query as shown in the following figure:



- This scan generates many I/O operations and utilizes many resources.

Finding Rows 2-2

- Following code snippet demonstrates how to create a table **Employee_Details** without an index:

```
USE CUST_DB
CREATE TABLE Employee_Details
(
EmpID int not null,
FirstName varchar(20) not null,
LastName varchar(20) not null,
DateofBirth datetime not null,
Gender varchar(6) not null,
City varchar(30) not null,
)
GO
```

- Assume that multiple records are inserted in the table, **Employee_Details**.
- The **SELECT** statement is used to search for records having the **FirstName** as John.
- Since there is no index associated with the **FirstName** column, SQL Server will perform a full table scan.

Finding Rows with Nonclustered Index 1-3

A nonclustered index is similar to a book index; the data and the index are stored in different places.

The pointers in the leaf level of the index point to the storage location of the data in the underlying table.

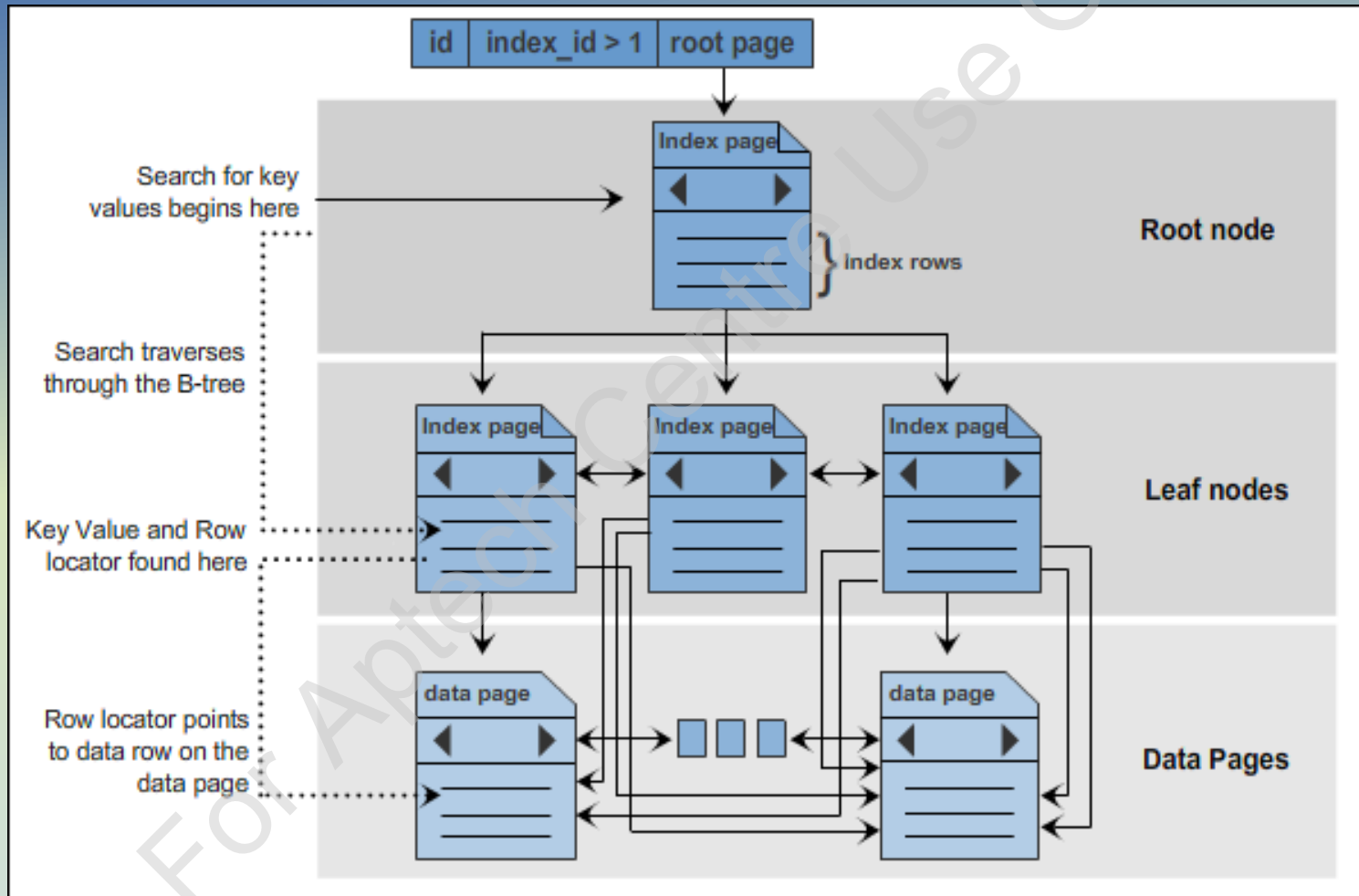
The nonclustered index is used to search for exact-match queries.

This is because the index contains entries describing the exact location of the data in the table.

For finding rows using nonclustered indexes, a `SELECT` statement is used with the nonclustered index column specified in the `WHERE` clause.

Finding Rows with Nonclustered Index 2-3

- Following figure shows the process of finding rows with nonclustered index:



Finding Rows with Nonclustered Index 3-3

- Following code snippet demonstrates how to create a nonclustered index **IX_EmployeeCity** on the **City** column of the **Employee_Details** table:

```
USE CUST_DB
CREATE NONCLUSTERED INDEX IX_EmployeeCity ON Employee_Details(City);
GO
```

- Assume that multiple records are inserted in the table, **Employee_Details**.
- The **SELECT** statement is used to search for records of employees from city, Boston as shown in the following code snippet:

```
USE CUST_DB
SELECT EmpID,FirstName,LastName,City FROM Employee_Details WHERE
City='Boston'
GO
```

- Since there is a nonclustered index associated with **City** column, SQL Server will use the **IX_EmployeeCity** index to extract the records as shown in the following figure:

| Results | | Messages | | |
|---------|-------|-----------|-----------|--------|
| | EmpID | FirstName | LastName | City |
| 1 | 101 | Andrew | Waller | Boston |
| 2 | 103 | Sophia | broderich | Boston |

Finding Rows in a Clustered Index 1-3

Clustered indexes store the data rows in the table based on their key values.

This data is stored in a sorted manner.

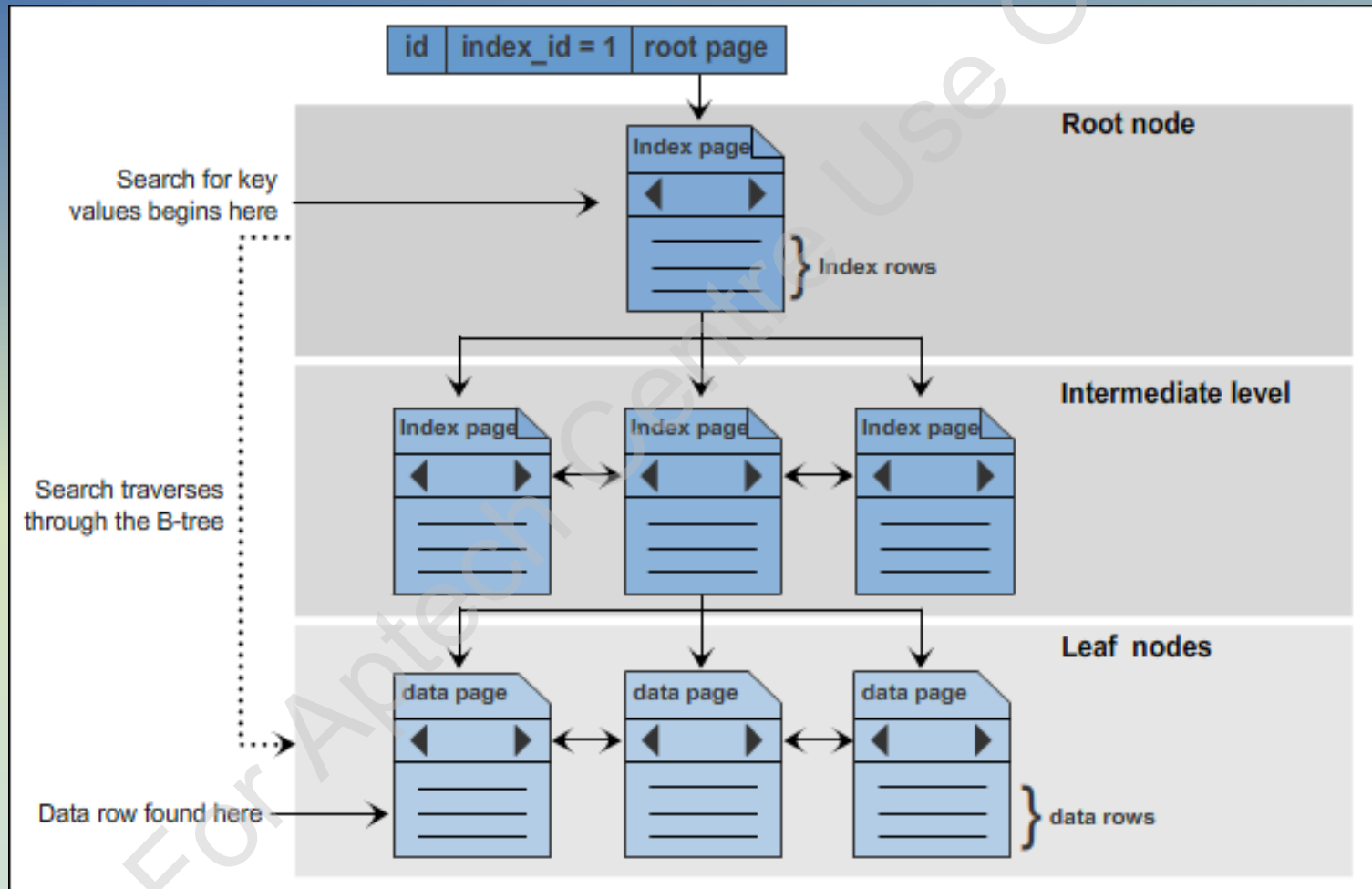
If the clustered key value is small, more number of index rows can be placed on an index page.

This decreases the number of levels in the index B-Tree that must be traversed to reach the data rows producing faster query results. This minimizes I/O overhead.

For finding rows using clustered indexes, a `SELECT` statement is used with the clustered index column specified in the `WHERE` clause.

Finding Rows in a Clustered Index 2-3

- Following figure shows the process of finding rows with clustered index:



Finding Rows in a Clustered Index 3-3

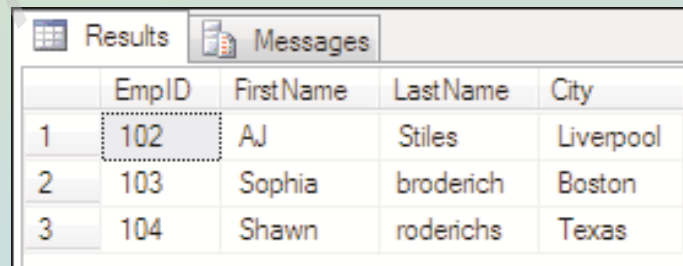
- Following code snippet demonstrates how to create a clustered index **IX_EmployeeID** on the **EmpID** column of the **Employee_Details** table:

```
USE CUST_DB
CREATE UNIQUE CLUSTERED INDEX IX_EmployeeID ON
Employee_Details (EmpID);
GO
```

- Assume that multiple records are inserted in the table, **Employee_Details**.
- The **SELECT** statement is used to search for records of employees having **EmpID** between 102 and 105 as shown in the following code snippet:

```
USE CUST_DB
SELECT EmpID, FirstName, LastName, City FROM Employee_Details WHERE EmpID
>= 102 AND EmpID <= 105;
GO
```

- Since there is a clustered index associated with the **EmpID** column, SQL Server will use the **IX_EmployeeID** index to extract the records as shown in the following figure:



| | EmpID | FirstName | LastName | City |
|---|-------|-----------|-----------|-----------|
| 1 | 102 | AJ | Stiles | Liverpool |
| 2 | 103 | Sophia | broderich | Boston |
| 3 | 104 | Shawn | roderichs | Texas |

Creating Unique Indexes 1-2

- A unique index can be created on a column that does not have any duplicate values.
- Once a unique index is created, duplicate values will not be accepted in the column.
- Thus, unique indexes should be created only on columns where uniqueness of values is a key characteristic. A unique index ensures entity integrity in a table.
- If a table definition has a `PRIMARY KEY` or a column with a `UNIQUE` constraint, SQL Server automatically creates a unique index when you execute the `CREATE TABLE` statement as shown in the following figure:

The diagram shows a table named **Customer_Details** with two columns: **CustID** and **FirstName**. The **CustID** column contains values 1 through 8, all of which are unique. The **FirstName** column contains the values John, Sam, Julie, Robert, John, George, Robert, and Merry. The first 'John' (row 1) is pointed to by an arrow labeled 'Unique Values'. The subsequent 'John' (row 5), 'Robert' (row 4), and 'Robert' (row 7) are pointed to by arrows from a single point labeled 'Duplicate Values', indicating that these values are repeated in the column.

| Customer_Details | |
|------------------|-----------|
| CustID | FirstName |
| 1 | John |
| 2 | Sam |
| 3 | Julie |
| 4 | Robert |
| 5 | John |
| 6 | George |
| 7 | Robert |
| 8 | Merry |

Creating Unique Indexes 2-2

- Unique index can be created using either the CREATE UNIQUE INDEX statement or using SSMS.
- The syntax used to create a unique index is as follows:

Syntax:

```
CREATE UNIQUE INDEX <index_name> ON <table_name>(<column_name>)
```

where,

column_name: specifies the name of the column on which the index is to be created.

UNIQUE: specifies that no duplicate values are allowed in the column.

- Following code snippet creates a unique index on the **CustID** column in the **Customer_Details** table:

```
CREATE UNIQUE INDEX IX_CustID ON Customer_Details(CustID)
```

Creating Computed Columns 1-2

A computed column is a virtual column in a table whose value is calculated at run-time.

The values in the column are not stored in the table but are computed based on the expression that defines the column.

The expression consists of a non-computed column name associated with a constant, a function, or a variable using arithmetical or logical operators.

- A computed column can be created using the `CREATE TABLE` or `ALTER TABLE` statements as shown in the following figure:

| Length | Breadth | Area |
|--------|---------|-------|
| 34 | 10 | 340 |
| 20 | 20 | 400 |
| 33.4 | 12 | 400.8 |
| 12 | 7 | 84 |

←
Computed column
(A=L*B)

Creating Computed Columns 2-2

- The syntax used to create a computed column is as follows:

Syntax:

```
CREATE TABLE <table_name> ([<column_name> AS  
<computed_column_expression>])
```

where,

table_name: Specifies the name of the table.

column_name AS computed_column_expression: Specifies the name of the computed column as well as the expression that defines the values in the column.

- Following code snippet creates a computed column **Area** whose values are calculated from the values entered in the **Length** and **Breadth** fields:

```
USE SampleDB  
CREATE TABLE Calc_Area (Length int, Breadth int, Area AS  
Length*Breadth)  
GO
```

Creating Index on Computed Columns 1-2

- An index can be created on a computed column if the column is marked `PERSISTED`.
- This ensures that the Database Engine stores computed values in the table.
- These values are updated when any other columns on which the computed column depends are updated.
- The database engine uses this persisted value when it creates an index on the column.
- An index is created on the computed column using the `CREATE INDEX` statement as shown in the following figure:

Computed column
(A=L*B)

| Length | Breadth | Area |
|--------|---------|-------|
| 34 | 10 | 340 |
| 20 | 20 | 400 |
| 33.4 | 12 | 400.8 |
| 12 | 7 | 84 |

| IX_Area |
|---------|
| 340 |
| 400 |
| 400.8 |
| 84 |

Creating Index on Computed Columns 2-2

- The syntax used to an index on a computed column is as follows:

Syntax:

```
CREATE INDEX <index_name> ON <table_name> (<computed_column_name>)
```

where,

computed_column_name: specifies the name of the computed column.

- Following code snippet creates an index **IX_Area** on the computed column **Area**:

```
USE SampleDB
CREATE INDEX IX_Area ON Calc_Area(Area);
GO
```


Cursors 1-7

- A database object that is used to retrieve data as one row at a time, from a resultset is called as cursors.
- Cursors are used when records in a database table need to be updated one row at a time.
- Types of Cursors:

Static Cursors

- These cursors help to populate the resultset when the cursor is created and the query result is cached. This is the slowest of all the cursors.
- This cursor can move/scroll in both backward and forward directions.
- Also, the static cursor cannot be updated or deleted.

Dynamic Cursors

- These cursors allow you to view the procedures of insertion, updation, and deletion when the cursor is open.
- This is one of the most sensitive cursor and is scrollable.

Forward Only Cursors

- These cursors also support updation and deletion of data.
- This is the fastest cursor though it does not support backward scrolling.
- The three types of forward cursors are `FORWARD_ONLY KEYSET`, `FORWARD_ONLY STATIC`, and `FAST_FORWARD`.

Keyset Driven Cursors

- These cursors create a set of unique identifiers as keys in a keyset that are used to control the cursor.
- This is also a sensitive cursor that can update and delete data.
- The keyset is dependent on all the rows that qualify the `SELECT` statement at the time of opening the cursor.

➤ The syntax used to declare a cursor is as follows:

Syntax:

```
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]  
[ FORWARD_ONLY | SCROLL ]  
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]  
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]  
[ TYPE_WARNING ]  
FOR select_statement  
[ FOR UPDATE [ OF column_name [ ,...n ] ] ]  
[;]
```

where,

`cursor_name`: is the name of the cursor defined, `cursor_name` must comply to the rules for identifiers.

Cursors 3-7

LOCAL: specifies that the cursor can be used locally to the batch, stored procedure, or trigger in which the cursor was created.

GLOBAL: specifies that the cursor can be used globally to the connection.

FORWARD_ONLY: specifies that the cursor can only be scrolled from the first to the last row.

STATIC: defines a cursor that makes a temporary copy of the data to be used by the cursor.

KEYSET: specifies that the membership and order of rows in the cursor are fixed when the cursor is opened.

DYNAMIC: defines a cursor that reflects all data changes made to the rows in its resultset as you scroll around the cursor.

FAST_FORWARD: specifies a **FORWARD_ONLY**, **READ_ONLY** cursor with performance optimizations enabled.

READ_ONLY: prevents updates made through this cursor.

SCROLL_LOCKS: specifies that positioned updates or deletes made through the cursor are guaranteed to succeed.

Cursors 4-7

- Following code snippet creates an **Employee** table in **SampleDB** database:

```
USE SampleDB
CREATE TABLE Employee
(
  EmpID int PRIMARY KEY,
  EmpName varchar (50) NOT NULL,
  Salary int NOT NULL,
  Address varchar (200) NOT NULL,
)
GO

INSERT INTO Employee(EmpID,EmpName,Salary,Address)
VALUES (1,'Derek',12000,'Houston')

INSERT INTO Employee(EmpID,EmpName,Salary,Address)
VALUES (2,'David',25000,'Texas')

INSERT INTO Employee(EmpID,EmpName,Salary,Address)
VALUES (3,'Alan',22000,'New York')

INSERT INTO Employee(EmpID,EmpName,Salary,Address)
VALUES (4,'Mathew',22000,'las Vegas')

INSERT INTO Employee(EmpID,EmpName,Salary,Address)
VALUES (5,'Joseph',28000,'Chicago')

GO

SELECT * FROM Employee
```

Cursors 5-7

- Following figure shows the output of the code:

| | EmpId | FirstName | Salary | City |
|---|-------|-----------|--------|-----------|
| 1 | 1 | Derek | 12000 | Houston |
| 2 | 2 | David | 25000 | Texas |
| 3 | 3 | Alan | 22000 | New York |
| 4 | 4 | Mathew | 22000 | Las Vegas |
| 5 | 5 | Joseph | 28000 | Chicago |

- Following code snippet demonstrates how to declare a cursor on **Employee** table:

```
USE SampleDB
SET NOCOUNT ON
DECLARE @Id int
DECLARE @name varchar(50)
DECLARE @salary int
/* A cursor is declared by defining the SQL statement that returns a
resultset.*/
DECLARE cur_emp CURSOR
```

Cursors 6-7

```
STATIC FOR
SELECT EmpID,EmpName,Salary from Employee
/*A Cursor is opened and populated by executing the SQL statement
defined by the cursor.*/
OPEN cur_emp
IF @@CURSOR_ROWS > 0
BEGIN
/*Rows are fetched from the cursor one by one or in a block to do data
manipulation*/
FETCH NEXT FROM cur_emp INTO @Id,@name,@salary
WHILE @@Fetch_status = 0
BEGIN
PRINT 'ID : '+ convert(varchar(20),@Id)+' , Name : '+@name+ ' , Salary :
'+convert(varchar(20),@salary)
FETCH NEXT FROM cur_emp INTO @Id,@name,@salary
END
END
--close the cursor explicitly
CLOSE cur_emp
/*Delete the cursor definition and release all the system resources
associated with the cursor*/
DEALLOCATE cur_emp
SET NOCOUNT OFF
```

Cursors 7-7

- In the code, the details are retrieved one row at a time.
- This procedure will help in retrieving large databases sequentially.
- First, a cursor is declared by defining the SQL statement that returns a resultset.
- Then, it is opened and populated by executing the SQL statement defined by the cursor.
- Rows are then fetched from the cursor one by one or in a block to perform data manipulation.
- The cursor is then closed and finally, the cursor definition is deleted and all the system resources associated with the cursor are released.
- Following figure shows the output of the code:

| Messages | | |
|----------|----------------|----------------|
| ID : 1, | Name : Derek, | Salary : 12000 |
| ID : 2, | Name : David, | Salary : 25000 |
| ID : 3, | Name : Alan, | Salary : 22000 |
| ID : 4, | Name : Mathew, | Salary : 22000 |
| ID : 5, | Name : Joseph, | Salary : 28000 |

Summary

- Indexes increase the speed of the querying process by providing quick access to rows or columns in a data table.
- SQL Server 2012 stores data in storage units known as data pages.
- All input and output operations in a database are performed at the page level.
- SQL Server uses catalog views to find rows when an index is not created on a table.
- A clustered index causes records to be physically stored in a sorted or sequential order.
- A nonclustered index is defined on a table that has data either in a clustered structure or a heap.
- XML indexes can speed up queries on tables that have XML data.
- Column Store Index enhances performance of data warehouse queries extensively.