Microsoft

SQL Server Index Structure

Module 5

Learning Units covered in this Module

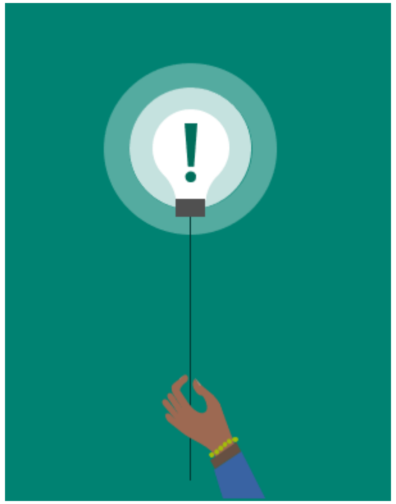
- Lesson 1: Index Internals
- Lesson 2: Index Strategy
- Lesson 3: Index Monitoring and Fragmentation

Lesson 1: Index Internals

Objectives

After completing this learning, you will be able to:

- Understand differences between heap and clustered objects.
- Describe B-Trees, and why they are beneficial.
- Understand the differences between clustered and non-clustered indexes.
- Describe the different methods by which data structures are accessed.



What is an Index?

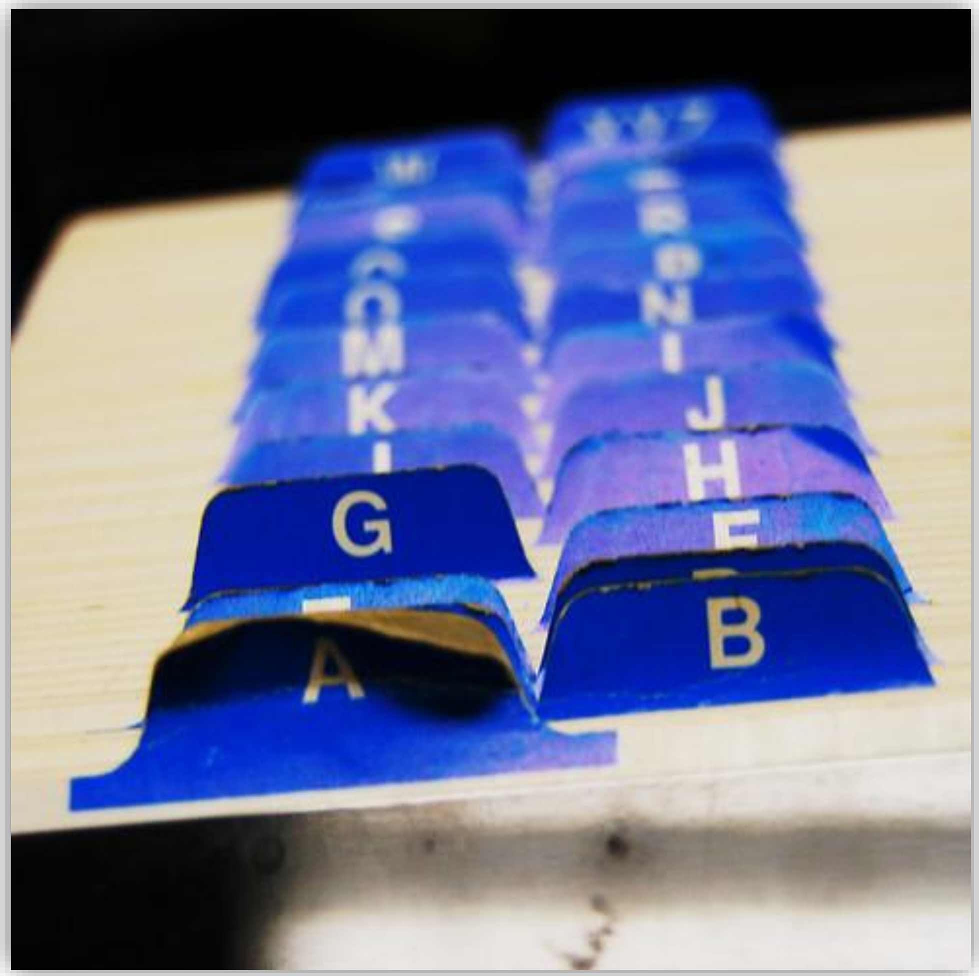
An index is an on-disk structure associated with a table that speeds retrieval of rows.

An index contains keys built from one or more columns in the table.

What is SARGability?

A SARGable item in a search predicate is able to use an index.

Non-SARGable expressions can significantly slow down queries.



How Data is Stored in Data Pages

Data stored in a Heap is not stored in any order and normally does not have a Primary Key.

Heap

AcctID	AcctName	Balance
1	Jack	500.00
2	Diane	750.00
29	Kelli	1250.00
27	Jessica	1005.00
18	Maddison	745.00
22	Bella	445.00

Clustered Index data is stored in sorted order by the Clustering key. In many cases, this is the same value as the Primary Key.

Clustered Index

AcctID	AcctName	Balance
1	Jack	500.00
2	Diane	750.00
12	Danny	630.00
14	Mayleigh	204.00
15	Molly	790.00
18	Maddison	745.00

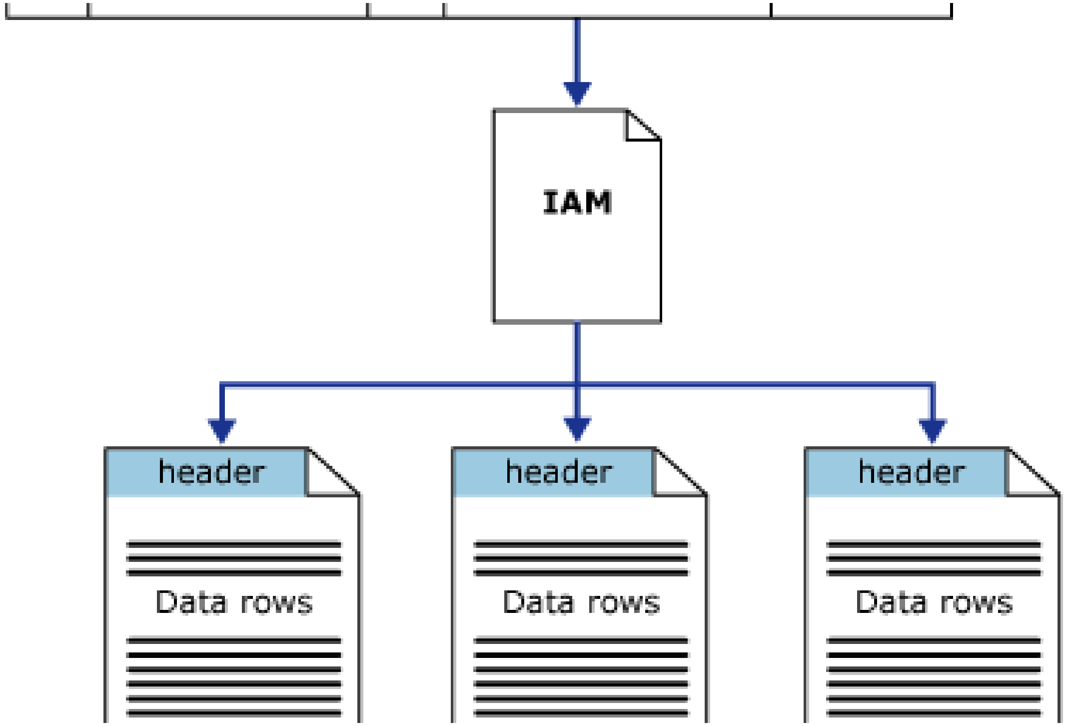
Heap

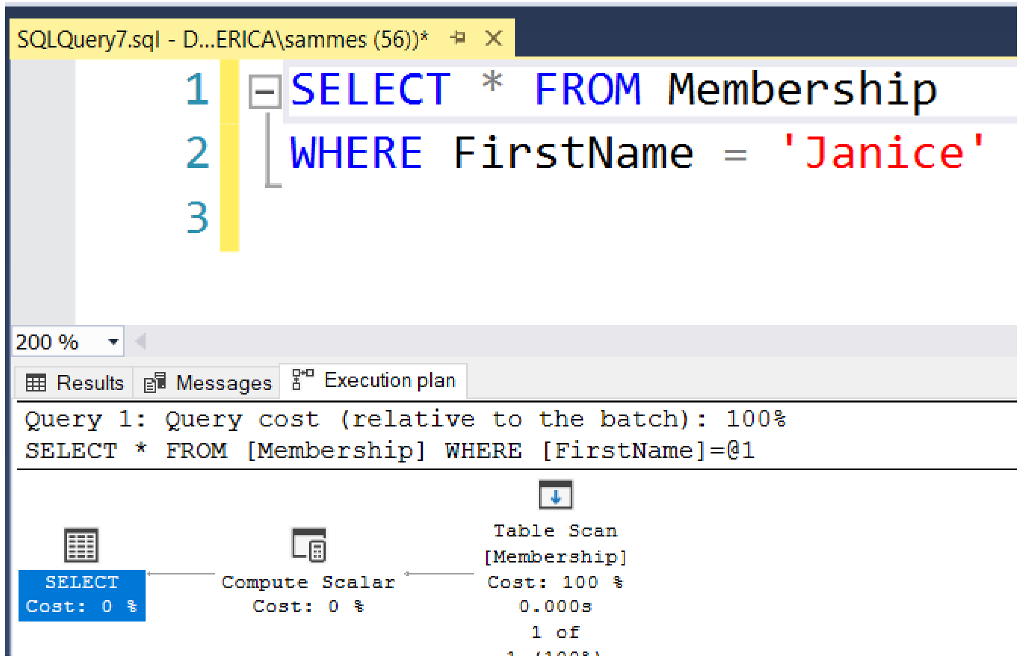
A heap is a table without a clustered index

- Unordered masses of data
- Can be good for quickly importing large sets of data
- Not a good idea for reporting-based data structures
- Use the ALTER TABLE…REBUILD command to “rebuild” a heap table
- Do not use a heap
  - When the data is frequently returned in a sorted order.
  - When the data is frequently grouped together.
  - When ranges of data are frequently queried from the table.
  - When there are no nonclustered indexes and the table is large.

Heap Structures

Heaps have one row in sys.partitions, with index_id = 0 for each partition used by the heap





Clustered Indexes

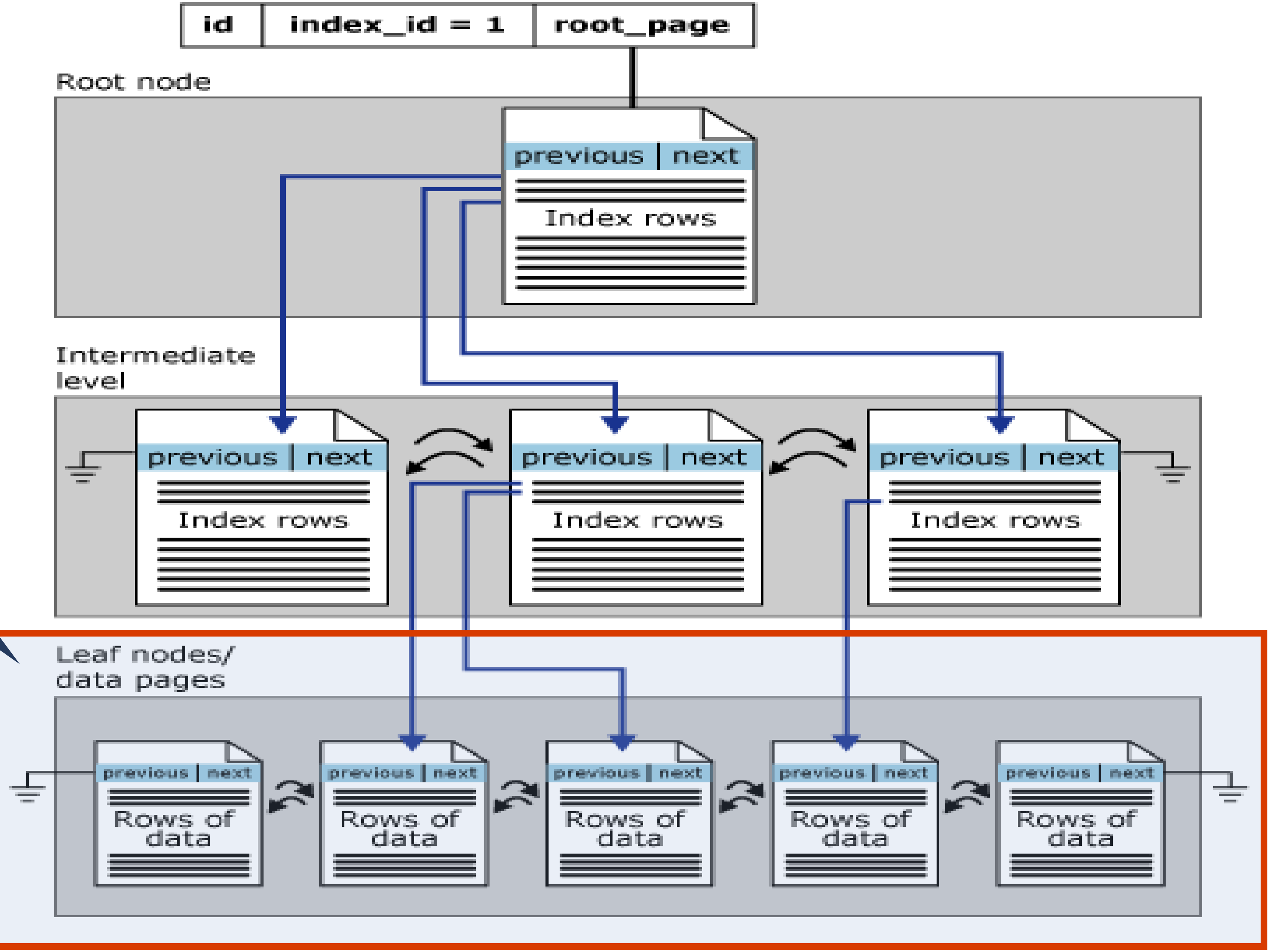
An ordered data structure that is implemented as a Balanced (B) Tree.

When a table is clustered, the leaf level of the index contains ALL data in the table. This means that the clustered index IS the table. This is also why there is only one per table.

The leaf level of the clustered index contains data pages.

Clustered Index Structure

Leaf Node Contains Data Pages



Demonstration

Examining index metadata and Building a B-Tree

- This demonstration reviews index, partition, and allocation unit metadata.
- Show how a B-Tree begins to form as rows are added to a table.

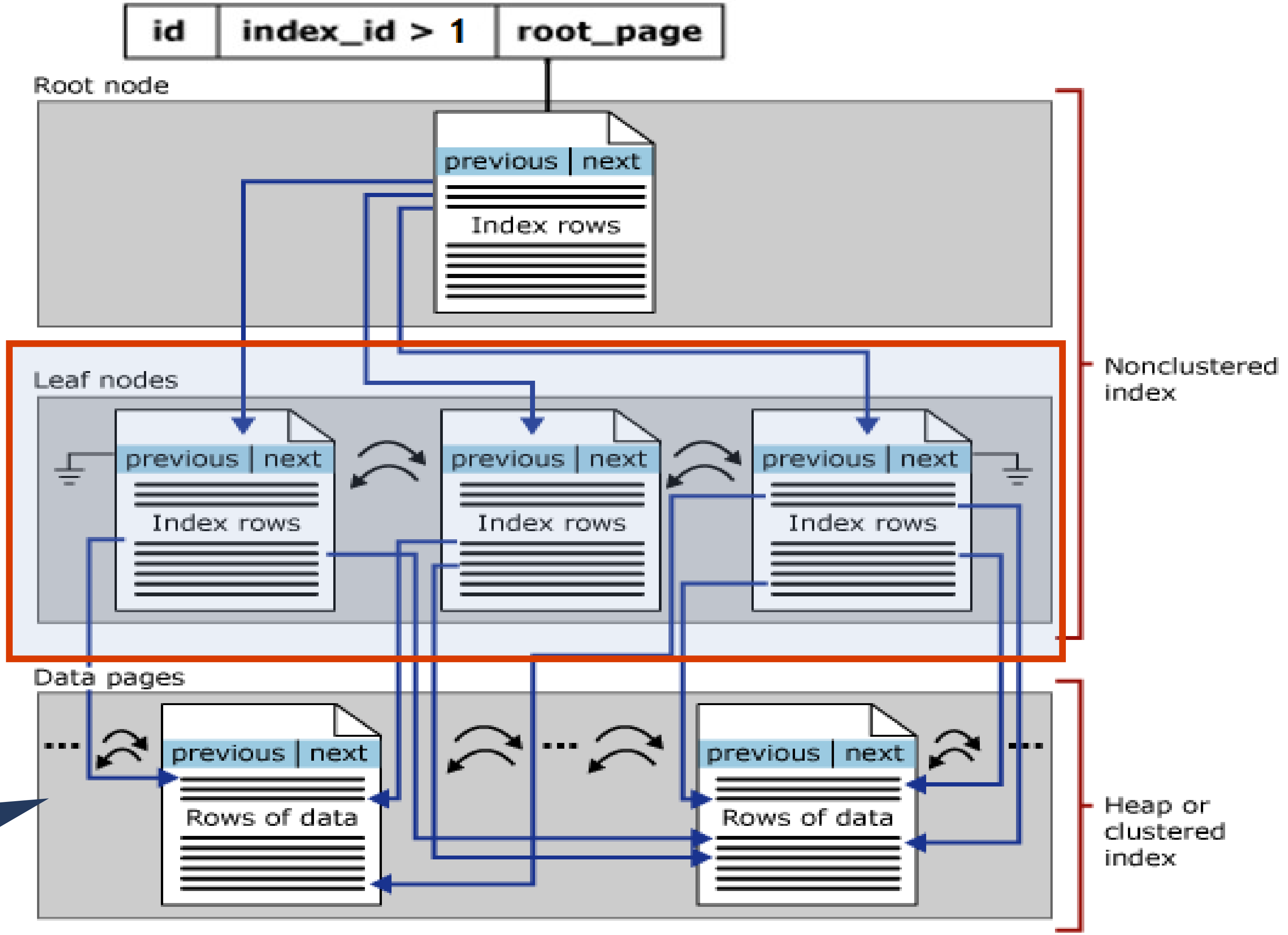
Non-Clustered Indexes

- Same B-tree data structure as a clustered index.
- It is a separate structure built on top of a Heap or Clustered Index.
- Only contains a subset of the columns in the base table
- The leaf level contains only the columns defined in the index as well as the clustered key/heap ID that points to the base table structure.

Non-Clustered Index Structure

Leaf Node has Pointers to Base Table

Base Table (Heap or Clustered Index)



Non-Clustered Index Structure

StateProvinceID (key)	AddressID (key)
3	912
3	929
3	932
3	940
3	944
3	951
3	18752

Non-Clustered Index
Leaf Level

ChildPageId	AddressID (key)
11854	804
11855	862
11856	920
11857	976

Clustered Index
Non-Leaf Level

```
Slot 50 Column 1 Offset
AddressID = 912
Slot 50 Column 2 Offset
AddressLine1 = 2525 Elec
Slot 50 Column 3 Offset
AddressLine2 = [NULL]
Slot 50 Column 4 Offset
City = Huntsville
Slot 50 Column 5 Offset
StateProvinceID = 3
```

Data Page 11855
Leaf Level

```
--SELECT * will need to find all columns
SELECT *
FROM Person.Address
WHERE StateProvinceID = 3
```

Clustered vs Nonclustered Indexes

An index is an on-disk structure associated with a table or view that speeds retrieval of rows.

Clustered Indexes

- Defines the order in which data is physically stored in a table.
- Table data can be sorted in only one way.
- Leaf level has data rows stored with index.
- When a table has a clustered index, the object is called clustered table.
- Cluster key is added to nonclustered index (as the pointer), keep it as narrow as possible.

Non-clustered Indexes

- Separate structure from base table.
- Contains a pointer back to base table called:
  - Row ID (RID when base table is HEAP)
  - Key (KEY when base table is Clustered)
- "Skinny" data structure as it contains a subset of base table only.
- To by-pass index key limits (1,700 bytes) ,non-key columns can be added to leaf level.
- As Leaf level contains fewer columns than base table, the non-clustered index uses fewer pages than the corresponding base table.

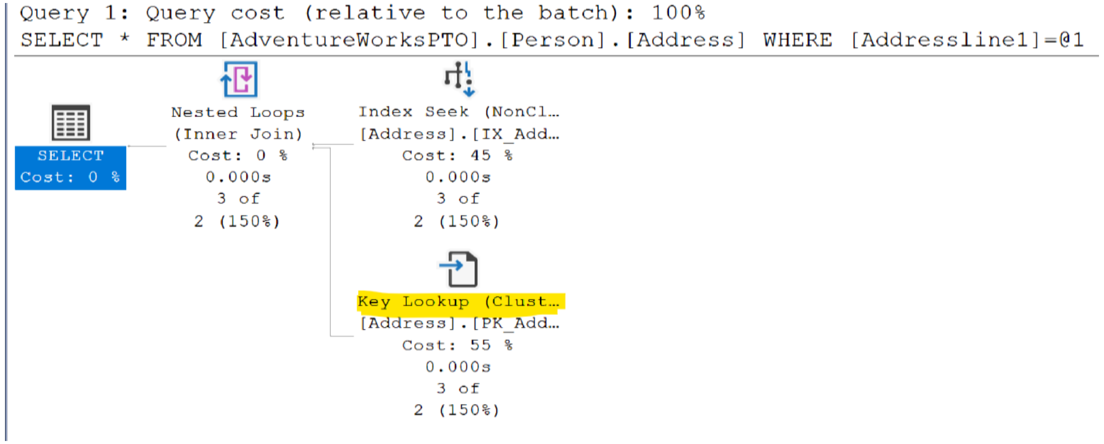
Bookmark Lookup or RID lookup

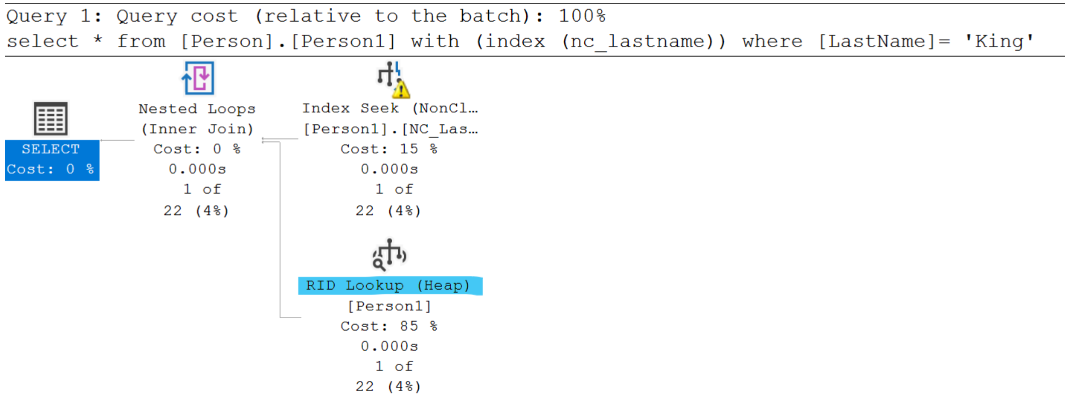
Occurs when SQL uses a nonclustered index to satisfy all or some of a query's predicates, but it doesn't contain all the information needed to cover the query.

Lookup effectively join the nonclustered index back to the clustered index or heap.

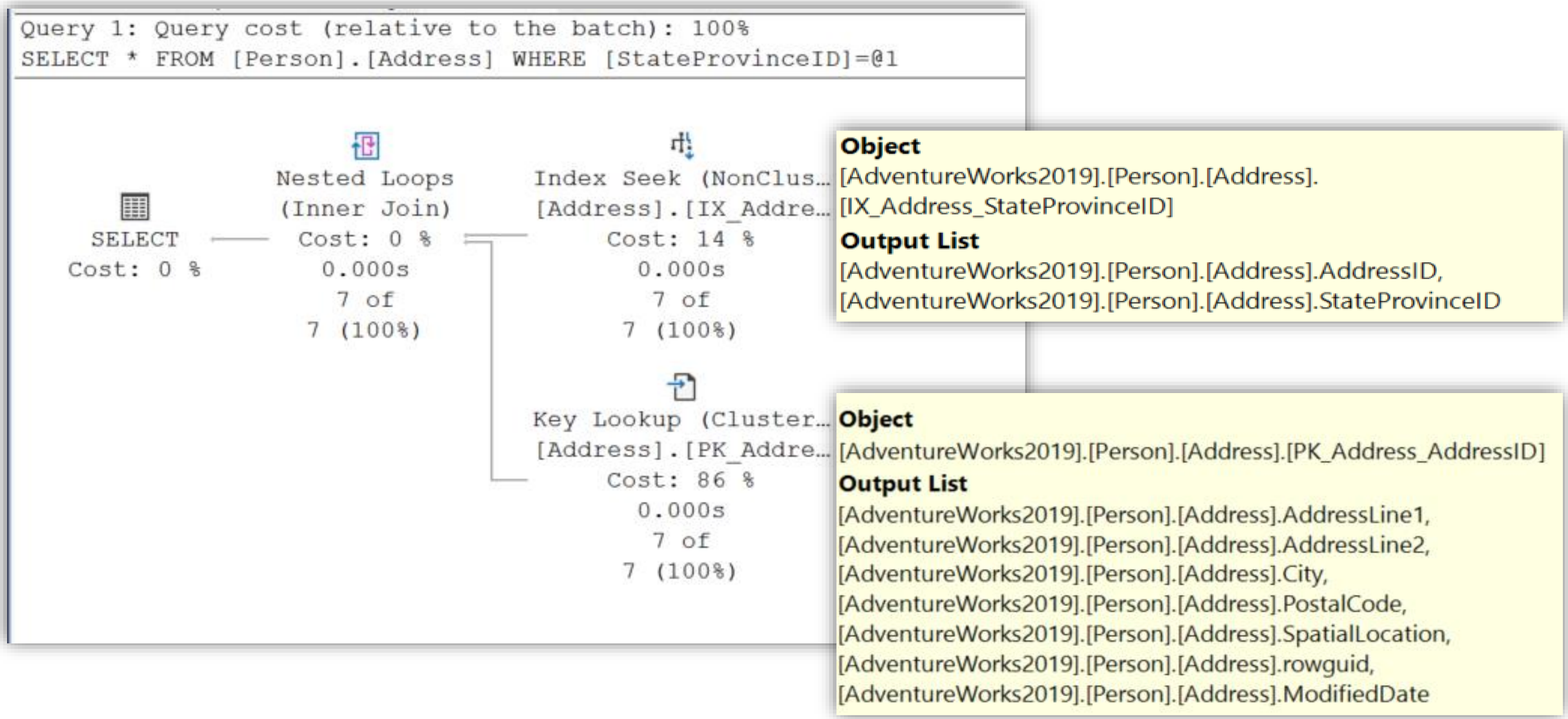
- If table has clustered index, it is called **bookmark lookup** (or key lookup)
- If the table is a heap with a non-clustered index, it is called **RID lookup**

This is an expensive operation.



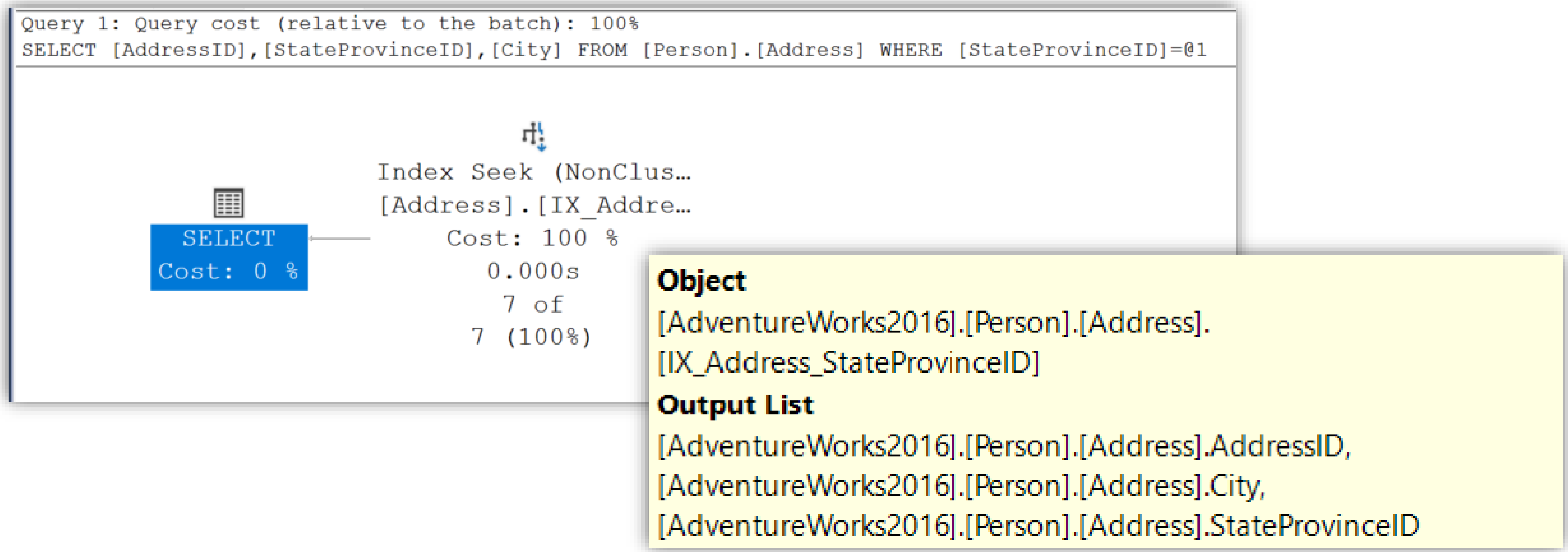


Key Lookup



Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [Person].[Address] WHERE [StateProvinceID]=@1
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
0.000s
7 of
7 (100%)
Index Seek (NonClus…
[Address].[IX_Addre…
Cost: 14 %
0.000s
7 of
7 (100%)
Key Lookup (Cluster…
[Address].[PK_Addre…
Cost: 86 %
0.000s
7 of
7 (100%)
Object
[AdventureWorks2019].[Person].[Address].
[IX_Address_StateProvinceID]
Output List
[AdventureWorks2019].[Person].[Address].AddressID,
[AdventureWorks2019].[Person].[Address].StateProvinceID
Object
[AdventureWorks2019].[Person].[Address].[PK_Address_AddressID]
Output List
[AdventureWorks2019].[Person].[Address].AddressLine1,
[AdventureWorks2019].[Person].[Address].AddressLine2,
[AdventureWorks2019].[Person].[Address].City,
[AdventureWorks2019].[Person].[Address].PostalCode,
[AdventureWorks2019].[Person].[Address].SpatialLocation,
[AdventureWorks2019].[Person].[Address].rowguid,
[AdventureWorks2019].[Person].[Address].ModifiedDate

Non-Clustered Index with Included Column



Single Vs Multi-column indexes

Single-column Indexing

- Good choice for columns that are highly selective, or columns referenced often by criteria.
- When filtering data from table with multiple single column indexes, SQL uses the index with higher cardinality column, it may be better to not index the low cardinality column.

Multi-Column Indexing

- Good choice for low cardinality columns.
- Highest cardinality field first, to zoom into a narrow range of the B-Tree as quickly as possible.
- Equality columns before inequality columns.

Multi-Column Indexing Access

Seek only happens if you search for the columns in the specified order.

CREATE INDEX IX1 ON TABLE (PostalCode, StateID, City)

Effect of column in different WHERE clause.

- WHERE PostalCode = 98011 – seek
- WHERE PostalCode = 98011 AND StateID = 79 – seek both
- WHERE PostalCode = 98011 – seek AND City = Bothell -- scan
- WHERE StateID = 79 -- scan

Multi-Column Indexing Access (Seek Predicates)

```sql
--Single value performs Index Seek.
SELECT City, StateProvinceID,
PostalCode
FROM Person.Address
WHERE PostalCode = '98011'
```

Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Object

[AdventureWorks2019].[Person].[Address].[IX_Postal_State_City]

Output List

[AdventureWorks2019].[Person].[Address].City,
[AdventureWorks2019].[Person].[Address].StateProvinceID,
[AdventureWorks2019].[Person].[Address].PostalCode

Seek Predicates

Seek Keys[1]: Prefix: [AdventureWorks2019].[Person].
[Address].PostalCode = Scalar Operator(CONVERT_IMPLICIT(nvarchar
(4000),[@1],0))

```sql
--Index Seek on both columns
--Search condition in same order as Index.
SELECT City, StateProvinceID, PostalCode
FROM Person.Address
WHERE PostalCode = '98011' AND
StateProvinceID = 79
```

Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Object

[AdventureWorks2019].[Person].[Address].[IX_Postal_State_City]

Output List

[AdventureWorks2019].[Person].[Address].City,
[AdventureWorks2019].[Person].[Address].StateProvinceID,
[AdventureWorks2019].[Person].[Address].PostalCode

Seek Predicates

Seek Keys[1]: Prefix: [AdventureWorks2019].[Person].
[Address].PostalCode, [AdventureWorks2019].[Person].
[Address].StateProvinceID = Scalar Operator(CONVERT_IMPLICIT
(nvarchar(4000),[@1],0)), Scalar Operator(CONVERT_IMPLICIT(int,
[@2],0))

Multi-Column Indexing Access (Scan Predicates)

```sql
--Index Seek on first column
--After seek, will scan second column
SELECT City, StateProvinceID, PostalCode
FROM Person.Address
WHERE PostalCode = '98011' AND City =
'Bothell'
```

```sql
--Search condition not in same order as
Index.
--Performs Index Scan.
SELECT  City, StateProvinceID, PostalCode
FROM Person.Address
WHERE StateProvinceID = 79
```

Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Predicate
[AdventureWorks2019].[Person].[Address].[City]=CONVERT_IMPLICIT(nvarchar(4000),[@2],0)

Object
[AdventureWorks2019].[Person].[Address].[IX_Postal_State_City]

Output List
[AdventureWorks2019].[Person].[Address].City,
[AdventureWorks2019].[Person].[Address].StateProvinceID,
[AdventureWorks2019].[Person].[Address].PostalCode

Seek Predicates
Seek Keys[1]: Prefix: [AdventureWorks2019].[Person].[Address].PostalCode = Scalar Operator(CONVERT_IMPLICIT(nvarchar(4000),[@1],0))

Index Scan (NonClustered)

Scan a nonclustered index, entirely or only a range.

Predicate
[AdventureWorks2019].[Person].[Address].[StateProvinceID]=(79)

Object
[AdventureWorks2019].[Person].[Address].[IX_Postal_State_City]

Output List
[AdventureWorks2019].[Person].[Address].City,
[AdventureWorks2019].[Person].[Address].StateProvinceID,
[AdventureWorks2019].[Person].[Address].PostalCode

Demonstration

Access Methods

Demonstrate seeks, scans, and lookups

Questions?

Knowledge Check

What are the three access methods reviewed in this lesson?

What are the three different types of allocations?

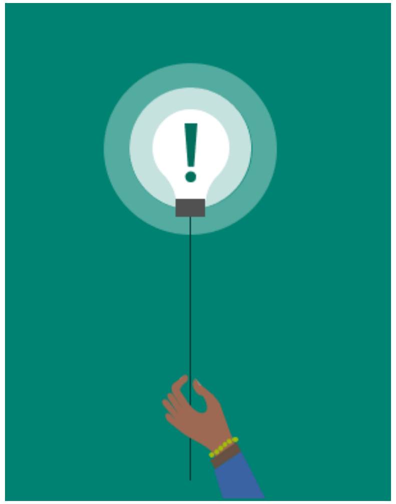
What is the primary goal of indexing?

Lesson 2: Index Strategy

Objectives

After completing this learning, you will be able to:

- Describe best practices associated with clustered index design.
- Consider factors involved in designing a nonclustered indexing strategy.
- Explain how an indexing strategy can reduce logical reads.
- Describe use of Filter indexes and Indexed views.
- Explain the role of Fill factor.
- Use a technique to reduce the fragmentation.
- Apply Index Compression.



Characteristics of a Good Clustering Key

Narrow	Unique	Static	Increasing
• Use a data type with a small number of bytes to conserve space in tables and indexes	• To avoid SQL adding a 4-byte uniquifier	• Allows data to stay constant without constant changes which could lead to page splits	• Allows better write performance and reduces fragmentation issues

Best Practices for Clustered indexes

The primary goal of any indexing strategy is to make queries faster

Best Practice	Seeks, not scans	Avoid key lookups	Get most rows on the index page	Get most rows on the data page	Avoid page splits
Use frequently searched value	X	X			
Use narrow keys			X	X	
Use unique values			X	X	
Use static values					X
Use ascending or descending key					X

Clustered Index vs Primary key considerations

Frequently, clustering on the primary key supports the clustered index best practices

- Query predicate use Primary Key
- Primary keys are involved in joins to foreign keys
- Primary keys are unique
- Primary keys are typically ascending
- Primary keys are typically not updated

Do not cluster on the Primary key IF:

- When table is accessed by other keys than the primary key
- Common issue accessing association or child table, when these tables require different primary key than parent table
- Use sys.dm_db_index_usage_stats to review data access patterns of index

Non-Clustered Index Strategy

Index columns used in WHERE clauses and JOINs.

Keep indexes to a minimum, to minimize impact on DML and log writes.

Two basic approaches: Single column indexes or Multicolumn indexes

Easy to create too many nonclustered indexes.

Specific indexes are appropriate for high-impact queries.

Ideally the focus should be writing queries to use existing indexes, rather than on adding more indexes.

Single Vs Multi-column indexes

Single-column Indexing

- Good choice for columns that are highly selective, or columns referenced often by criteria.
- When filtering data from table with multiple single column indexes, SQL uses the index with higher cardinality column, it may be better to not index the low cardinality column.

Multi-Column Indexing

- Good choice for low cardinality columns.
- Highest cardinality field first, to zoom into a narrow range of the B-Tree as quickly as possible.
- Equality columns before inequality columns.

Multi-Column Indexing Access

Seek only happens if you search for the columns in the specified order.

CREATE INDEX IX1 ON TABLE (PostalCode, StateID, City)

Effect of column in different WHERE clause.

- WHERE PostalCode = 98011 – seek
- WHERE PostalCode = 98011 AND StateID = 79 – seek both
- WHERE PostalCode = 98011 – seek AND City = Bothell -- scan
- WHERE StateID = 79 -- scan

Multi-Column Indexing Access (Seek Predicates)

```sql
--Single value performs Index Seek.
SELECT City, StateProvinceID,
PostalCode
FROM Person.Address
WHERE PostalCode = '98011'
```

```sql
--Index Seek on both columns
--Search condition in same order as Index.
SELECT City, StateProvinceID, PostalCode
FROM Person.Address
WHERE PostalCode = '98011' AND
StateProvinceID = 79
```

Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Object

[AdventureWorks2019].[Person].[Address].[IX_Postal_State_City]

Output List

[AdventureWorks2019].[Person].[Address].City,
[AdventureWorks2019].[Person].[Address].StateProvinceID,
[AdventureWorks2019].[Person].[Address].PostalCode

Seek Predicates

Seek Keys[1]: Prefix: [AdventureWorks2019].[Person].[Address].PostalCode = Scalar Operator(CONVERT_IMPLICIT(nvarchar(4000),[@1],0))

Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Object

[AdventureWorks2019].[Person].[Address].[IX_Postal_State_City]

Output List

[AdventureWorks2019].[Person].[Address].City,
[AdventureWorks2019].[Person].[Address].StateProvinceID,
[AdventureWorks2019].[Person].[Address].PostalCode

Seek Predicates

Seek Keys[1]: Prefix: [AdventureWorks2019].[Person].[Address].PostalCode, [AdventureWorks2019].[Person].[Address].StateProvinceID = Scalar Operator(CONVERT_IMPLICIT(nvarchar(4000),[@1],0)), Scalar Operator(CONVERT_IMPLICIT(int,[@2],0))

Multi-Column Indexing Access (Scan Predicates)

```sql
--Index Seek on first column
--After seek, will scan second column
SELECT City, StateProvinceID, PostalCode
FROM Person.Address
WHERE PostalCode = '98011' AND City =
'Bothell'
```

```sql
--Search condition not in same order as
Index.
--Performs Index Scan.
SELECT  City, StateProvinceID, PostalCode
FROM Person.Address
WHERE StateProvinceID = 79
```

Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Predicate
[AdventureWorks2019].[Person].[Address].[City]=CONVERT_IMPLICIT(nvarchar(4000),[@2],0)

Object
[AdventureWorks2019].[Person].[Address].[IX_Postal_State_City]

Output List
[AdventureWorks2019].[Person].[Address].City, [AdventureWorks2019].[Person].[Address].StateProvinceID, [AdventureWorks2019].[Person].[Address].PostalCode

Seek Predicates
Seek Keys[1]: Prefix: [AdventureWorks2019].[Person].[Address].PostalCode = Scalar Operator(CONVERT_IMPLICIT(nvarchar(4000),[@1],0))

Index Scan (NonClustered)

Scan a nonclustered index, entirely or only a range.

Predicate
[AdventureWorks2019].[Person].[Address].[StateProvinceID]=(79)

Object
[AdventureWorks2019].[Person].[Address].[IX_Postal_State_City]

Output List
[AdventureWorks2019].[Person].[Address].City, [AdventureWorks2019].[Person].[Address].StateProvinceID, [AdventureWorks2019].[Person].[Address].PostalCode

Demonstration

Analyzing Index Usage

Show how index usage patterns can help identify index and query improvements.

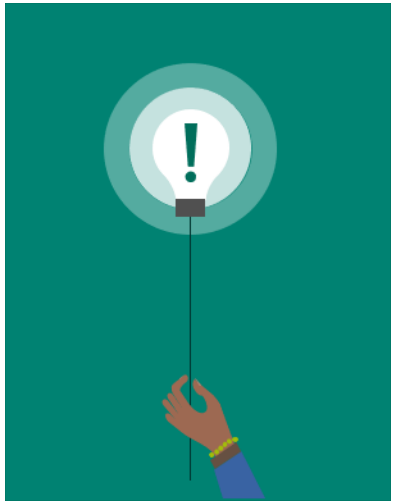
Questions?

Lesson 3: Index Monitoring and Fragmentation

Objectives

After completing this learning, you will be able to:

- Monitor index usage.
- Describe page splits.
- Understand problems caused by page splits.
- Monitor index fragmentation and free space.
- Review options to remove fragmentation and free space.



Why to monitor Index usage

Data access and distribution changes over time.

By reviewing index usage, we can ensure that the indexes are appropriate for the current workload.

Use sys.dm_db_index_usage_stats and sys.dm_db_index_operational_stats to monitor index usage.

Database standard report also monitors Index usage and Physical Statistics.

Clustered Index usage patterns

Pattern	Interpretation
Seeks, but no scans	• May not need maintenance. Seeks are not impacted by fragmentation
Clustered index with High number of scans	• Review missing index recommendations. • Consider changing Clustering key. • Simplify queries with complex WHERE clauses
Clustered index with Low number of seeks and high number of lookups	• May need to change Clustering Key • Look how often bookmark lookups occur.

Non-Clustered Index usage patterns

Pattern	Interpretation
Seeks, but no scans	• May not need maintenance. • Seeks are not impacted by fragmentation
Non-Clustered index with High number of scans	• Consider compression. • Check for non-SARGable queries. • Review missing index recommendations
Index with No/low usage	• May be able to drop or disable index
Non-Clustered Index Used at specific times only	• May be able to drop index, recreate when needed

sys.dm_db_index_usage_stats

- Tracks seeks, scans, and lookups
- Provides date of last access
- Counters are incremented once per query execution
- User and system access (such as index rebuilds) are tallied separately
- Counters are initialized to at SQL Server (MSSQLSERVER) service is restarts

```
SELECT
         Distinct OBJECT_NAME(Indx.OBJECT_ID) Table_Name  ,Indx.name  IndexName
         ,Indx.type_desc IndexType,Iusg.user_seeks  Seeks,Iusg.user_scans  Scans
         ,Iusg.user_lookups  Lookups ,Iusg.user_updates Updates ,Iusg.last_user_seek LastSeek
         ,Iusg.last_user_scan LastScan,Iusg.last_user_lookup LastLookup ,Iusg.last_user_update LastUpdate
FROM
         SYS.INDEXES Indx
         INNER JOIN SYS.DM_DB_INDEX_USAGE_STATS Iusg
                  ON Iusg.index_id = Indx.index_id AND Iusg.OBJECT_ID = Indx.OBJECT_ID
         INNER JOIN SYS.DM_DB_PARTITION_STATS PrtSts
                  ON PrtSts.object_id=Indx.object_id
WHERE
         OBJECTPROPERTY(Indx.OBJECT_ID,'IsUserTable') = 1
```

sys.dm_db_index_operational_stats

- Returns information about the lower-level I/O activities
- Takes database_id, the object_id, the index_id and the partition_number as parameters
- Memory-optimized indexes do not appear in this DMV

```
SELECT
       *
FROM
       SYS.DM_DB_INDEX_OPERATIONAL_STATS (NULL,NULL,NULL,NULL ) opsts
       INNER JOIN SYS.INDEXES AS Indx
              ON Indx.[OBJECT_ID] = opsts.[OBJECT_ID]
              AND Indx.INDEX_ID = opsts.INDEX_ID
WHERE
       OBJECTPROPERTY(opsts.[OBJECT_ID],'IsUserTable') = 1
```

sys.dm_db_index_physical_stats

- Returns size and fragmentation information for the data and indexes of the specified table or view in SQL Server.
- Takes database_id, the object_id, the index_id and the partition_number as parameters
- Memory-optimized indexes do not appear in this DMV

```sql
SELECT OBJECT_SCHEMA_NAME(I.object_id) AS SchemaName, OBJECT_NAME(I.object_id) AS
TableName, I.name, I.Index_ID, IPS.partition_number, IPS.avg_fragmentation_in_percent,
   IPS.page_count, IPS.avg_page_space_used_in_percent
FROM sys.indexes as I
INNER JOIN sys.dm_db_index_physical_stats(DB_ID(), NULL, NULL, NULL, 'Limited') as IPS
ON I.object_id = IPS. object_id
AND I.index_id = IPS.index_id
--WHERE IPS.avg_fragmentation_in_percent >30
--AND IPS.page_count >1000
```

Fragmentation

A fragmented table/Index is when some of its data pages point to pages that are not in sequence.

Logical fragmentation

- Occurs when leaf level pages are not physically corresponding to the logical order of the index:
  - Pages are not in the most efficient order for scanning purposes.
- Limits the efficiency of read-ahead scans, but not seeks.

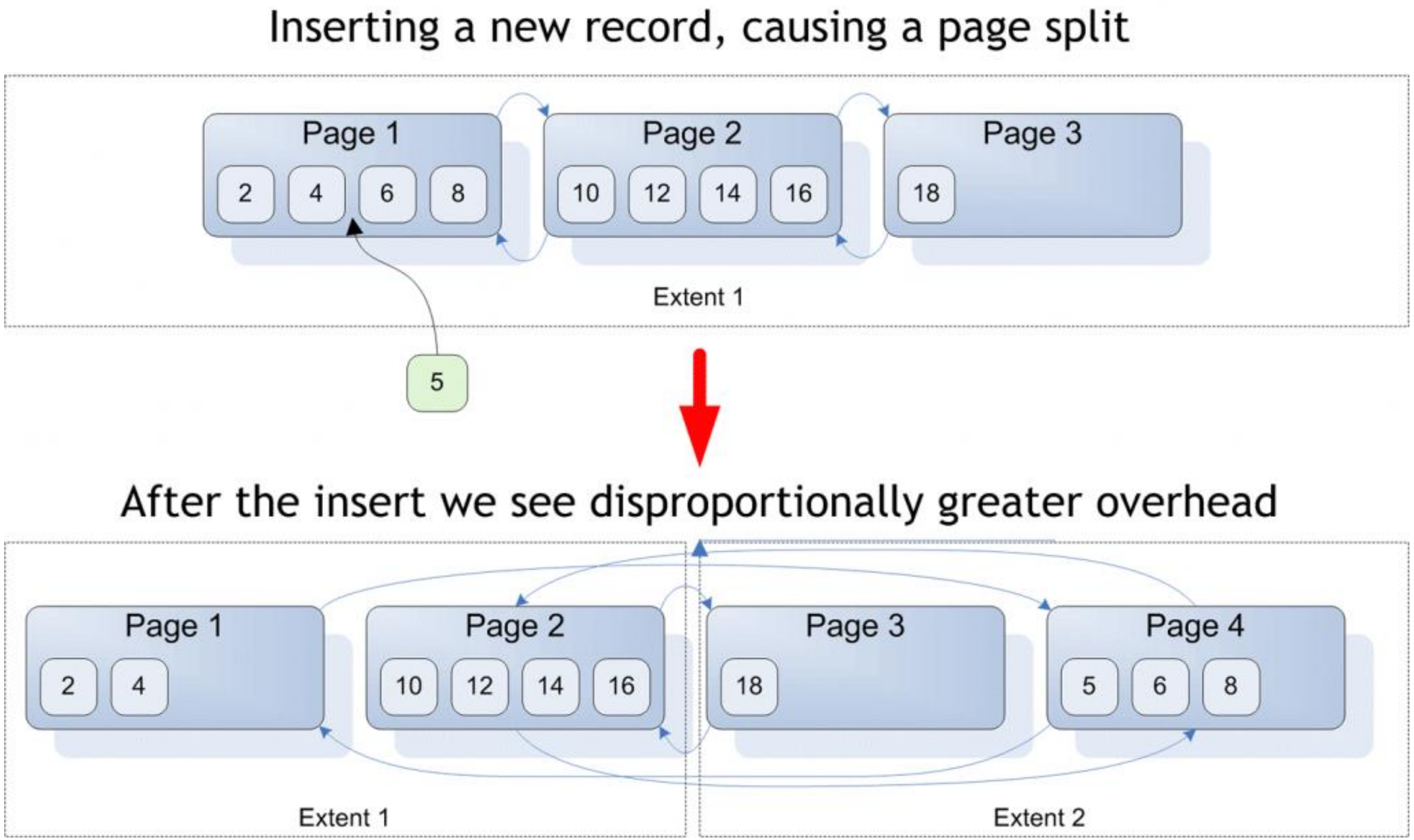
Page density

- How full a page is when a rebuild/reorganization occurs.
- The fuller a page is, the more likely page splits occur when data is modified.
- The less full a page is, the more wasted space in the buffer pool when reading pages.

Page splits

- Mechanism to optimize inserts and updates
- Occurs when page is full to hold a new or updated row
- Half the rows are moved to a new page
- The new page is added after the last existing page
- Cost associated with Page splits
  - Fragmentation, which affects scan performance
  - Wasted free space, which affects Page Density

Page Splits


Inserting a new record, causing a page split
Page 1
2
4
6
8
Page 2
10
12
14
16
Page 3
18
Extent 1
5
After the insert we see disproportionally greater overhead
Page 1
2
4
Page 2
10
12
14
16
Page 3
18
Page 4
5
6
8
Extent 1
Extent 2

SQL Index Fragmentation and Maintenance options

SSMS -> Index Physical Statistics report

- Will report fragmentation and recommend solution at database level

Custom Solution

- Find fragmentation using Sys.dm_db_index_physical_stats
- Correct fragmentation

Can remove Index fragmentation using Rebuild or Reorganize option

	ALTER INDEX… REBUILD	ALTER INDEX … REORGANIZE
Removes fragmentation	X	X
Removes free space	X	
Resets fill factor	X	
Updates statistics	X	

Fill Factor

Can address performance issues due to fragmentation.

Use When	Do not use when
• Specifying the amount of free space on a data or index page. • Reducing logical fragmentation.	• Low fragmentation • High seeks and low scans • Index is not scanned

```
ALTER INDEX [pk_bigProduct]
ON [dbo].[bigProduct]
REBUILD WITH (PAD_INDEX = ON, FILLFACTOR = 90)
```

Reducing fragmentation

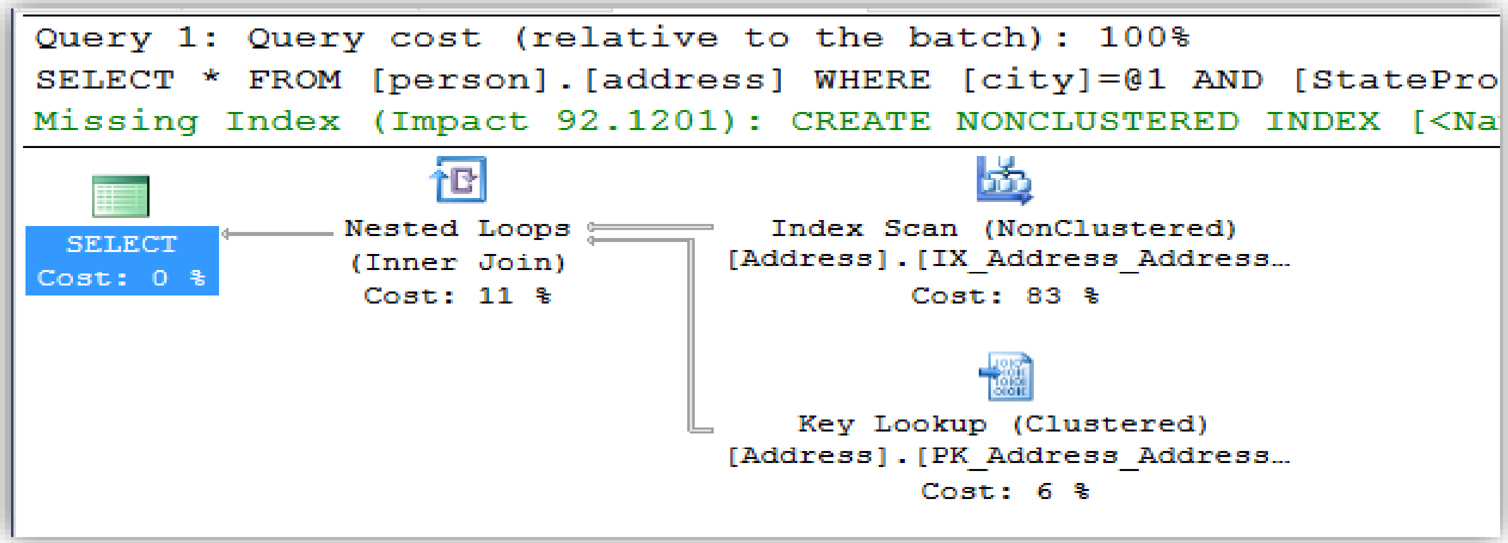
Reducing Fragmentation
• Using an ascending key (not always possible). • Using an appropriate fill factor for the workload. • Update in sets, rather than one at a time. • Do not insert with immediate update.

Missing Index Recommendations

SQL Server identifies indexes that would help a query's performance

The recommendation is included in the query plan

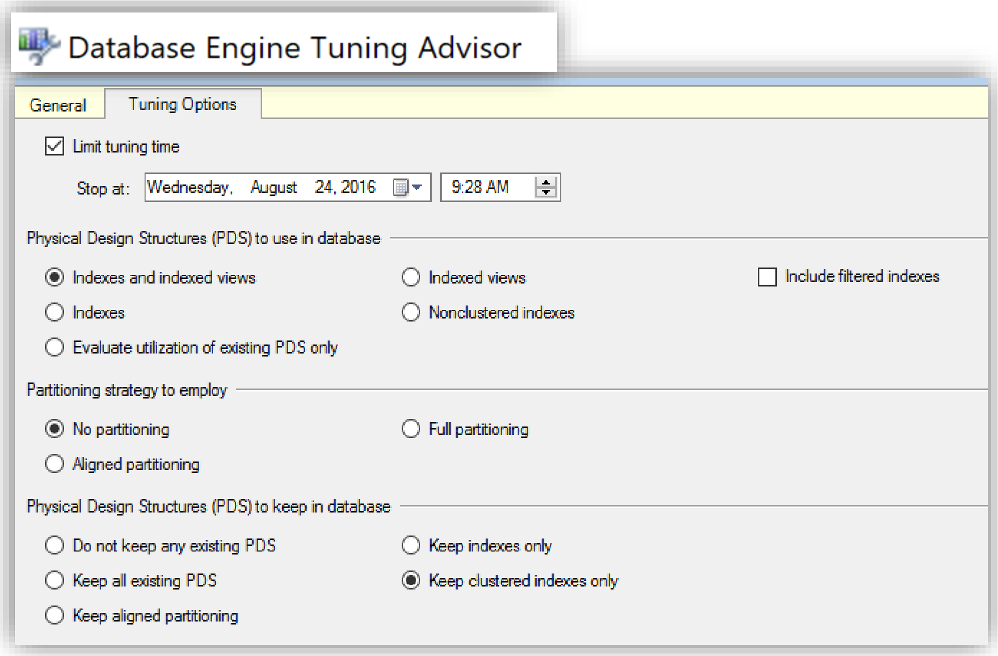
The cost savings are aggregated in DMVs to help identify the most beneficial indexes



Choosing Indexes

Following sources can help you choosing indexes

- Missing Index recommendations
- sys.dm_db_index_usage_stats
- Database Tuning Advisor



Demonstration

Analyzing Index Usage

Show how index usage patterns can help identify index and query improvements.

Index Maintenance

- Identifying and removing physical index fragmentation

LAB

Questions?

Knowledge Check

Which DMV/DMF are used to monitor index usage?

Does it make sense to add all SQL Server missing recommendations?

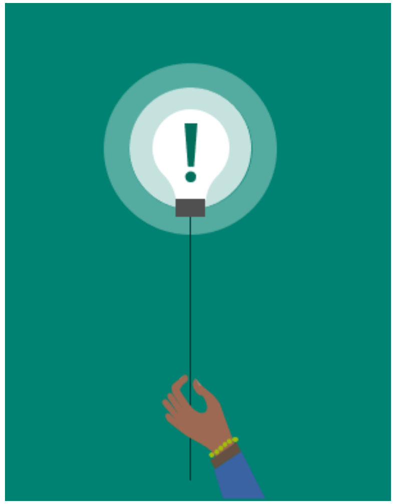
When Managed Lock Priority should be used?

Lesson 4: Partitioned Tables and Indexes

Objectives

After completing this learning, you will be able to:

- Understand Partition Concepts.
- Explain benefits of partitioning.
- Distinguish between aligned vs non-aligned indexes.
- Manage partitions.
- Apply Partition performance guidelines.



What is Partitioning?

"Horizontal Partition"

Table or Index Partition

- Breaking a single table or index into multiple HoBT (Heap or B-Tree) structures
- Requires a single column to use as partition key
- Partitioning is transparent to all queries

Partitioning benefits

Data Access

- Faster data load and offload.
- Query performance may be improved:
  - when predicate uses Partition-Key → partition elimination
  - when predicate doesn't use partition key → parallel execution

Maintenance and management

- Data placement handled by SQL Server.
- Piecemeal backup / restore of data by partition.
- Per-Partition management available for:
  - Index Maintenance
  - Lock Escalation
  - Compression

Horizontal Partitioning

Table without Partition

ProductKey	OrderDateKey	CustomerKey	SalesOrderNumber	UnitPrice	ExtendedAmount	ProductStandardCost
312	20050922	28191	SO44236	3578.27	3578.27	2171.2942
313	20050923	28163	SO44242	3578.27	3578.27	2171.2942
314	20050926	28142	SO44254	3578.27	3578.27	2171.2942
312	20050927	28171	SO44259	3578.27	3578.27	2171.2942

Partition Key

Table with Partition

Partition 1

ProductKey	OrderDateKey	CustomerKey	SalesOrderNumber	UnitPrice	ExtendedAmount	ProductStandardCost
312	20050922	28191	SO44236	3578.27	3578.27	2171.2942
313	20050923	28163	SO44242	3578.27	3578.27	2171.2942

Partition 2

ProductKey	OrderDateKey	CustomerKey	SalesOrderNumber	UnitPrice	ExtendedAmount	ProductStandardCost
314	20050926	28142	SO44254	3578.27	3578.27	2171.2942
312	20050927	28171	SO44259	3578.27	3578.27	2171.2942

Table Partitioning Concepts

Partition Function	Partition Schemes	Partition Column (key)	Partition Table & index
• Defines how the table is logically partitioned. • Defines data type of partition key, and ranges of each partition. • Ranges can be modified as needed.	• Specify mapping from partitioned table to Filegroup(s). • Uses *Partition Function* to perform mapping. • Different filegroups can provide increased performance, but is not required.	• Column used to apply partition function. • If using a computed column, it must be explicitly marked as PERSISTED.	• Partitioned Tables are defined using *Partition Schemes*. • Several tables may share common Partition Functions and Partition Schemes. • Loading data into the table or index creation, will move data to correct partition automatically.

Aligned vs Non-aligned indexes

Aligned Index

- An index that is built on the same partition scheme as its corresponding table.
- When a table and its indexes Align, SQL Server can switch partitions quickly and efficiently.

Non-aligned index

- An index partitioned independently from its corresponding table.
- Designing a non-aligned partitioned index can be useful in the following cases:
  - The base table has not been partitioned.
  - The index key is unique, and it does not contain the partitioning column of the table.
  - Useful when want to joins with more tables using different join columns.

Indexes on Partitioned Table

Logical Alignment

- Indexes are logically aligned when they use partitioning column.

Storage Alignment

- Indexes are storage aligned when created on the same partitioning scheme.
- By default, indexes will be created ON the partition scheme – ‘storage aligned’ indexes.

```
CREATE [CLUSTERED] INDEX … ON {partition_scheme_name ( column_name ) }
```

Partitioned table and index sample

```
-- Creating a partition function with four partitions
CREATE PARTITION FUNCTION myRangePF1 (int)
    AS RANGE LEFT FOR VALUES (1, 100, 1000) ;
GO
-- Creates a partition scheme that applies myRangePF1 to the four filegroups
CREATE PARTITION SCHEME myRangePS1
    AS PARTITION myRangePF1
    TO (test1fg, test2fg, test3fg, test4fg) ;
GO
-- Creates a partitioned table that uses myRangePS1 to partition col1
CREATE TABLE PartitionTable (col1 int NOT NULL, col2 char(10))
    ON myRangePS1 (col1) ;
GO
CREATE CLUSTERED INDEX ClusIndxCol1 ON PartitionTable(col1)
ON myRangePS1(col1)
GO
```

Table Partitioning Operations

Data Movement in table or Partition is done using ALTER …SWITCH/MERGE/SPLIT

SWITCH	MERGE	SPLIT
• Uses staging table • MUST have same schema as Partitioned table	• Used to eliminate an empty partition, by merging it with another partition	• Used to create a new empty partition from an existing partition

```
--SWITCH
ALTER TABLE PartitionTable SWITCH PARTITION 2 TO NonPartitionTable ;
--SPLIT
CREATE PARTITION FUNCTION myRangePF1 (int) AS RANGE LEFT FOR VALUES ( 1, 100, 1000 );
ALTER PARTITION FUNCTION myRangePF1() SPLIT RANGE (500);
--MERGE
CREATE PARTITION FUNCTION myRangePF1 (int) AS RANGE LEFT FOR VALUES ( 1, 100, 1000 );
ALTER PARTITION FUNCTION myRangePF1() MERGE RANGE (100);
```

Demonstration

Partitioning

Explore Partition Schemes and Functions.

SQL Server Partitioning

- Practicing with a sliding partition window

Questions?

Knowledge Check

What three structures are required to support partitions?

What are some benefits of partitioning?

What is used for data movement in partitions or table?

Microsoft