Indexes

Best Practices

S5

Indexes

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

- Great indexing → application fast & nimble
- Poor indexing → slows entire SQL Server

Index Characteristics

- Clustered versus nonclustered
- Unique versus nonunique
- Single column versus multicolumn
- Ascending or descending order on the columns in the index
- Full-table versus filtered for nonclustered indexes

Clustered vs NonClustered Index

- Clustered index: sorts and stores data rows in a table, based on search key values

```
CREATE CLUSTERED INDEX Index_Name
              ON Schema.TableName(Column);
```

- NonClustered index: key values and a pointer to data in the heap/clustered index

```
CREATE INDEX Index_Name
              ON Schema.TableName(Column);
```

Clustered vs NonClustered Index

- The data pages of a clustered index will always include *all columns* in the table

- There is only one clustered index per table.
- SQL Server supports up to 999 nonclustered indexes per table.
- An index key – clustered or nonclustered – can be a maximum of 16 columns and 900 bytes.

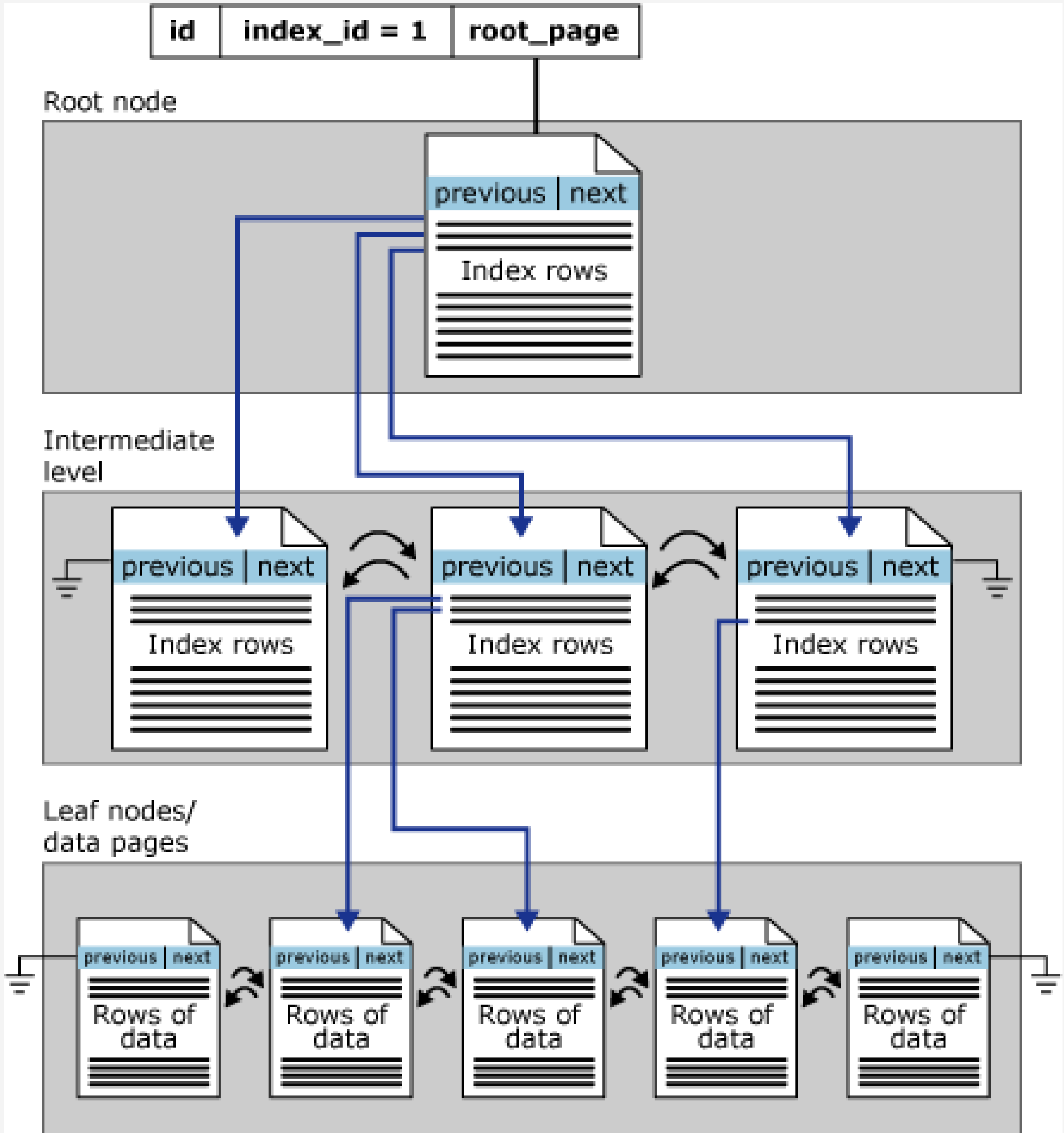
Clustered Index

- Can be used for frequently used queries
- Provide a high degree of uniqueness
- Can be used in range queries

- Clustered indexes are not a good choice for the following attributes:
  - Columns that undergo frequent changes
  - Wide keys

Clustered Index

- indexes are organized as B-trees.



Clustered vs NonClustered Index

- When you create a primary key on a table
  - + if a clustered index is not defined
  - + a nonclustered index is not specified
  - → a unique clustered index is created

- If all columns returned in a query are in the index: covering index

Key vs NonKey Index Columns

- Key columns: the columns specified to create an index.
- Nonkey columns: columns added to the INCLUDE clause of a nonclustered index.

```
CREATE INDEX Index_Name
ON Schema.TableName(Column)
INCLUDE (ColumnA, ColumnB);
```

Key vs NonKey Index Columns

- Benefits to using non-key columns
  - Columns can be accessed with an index scan.
  - Data types not allowed in key columns are allowed in nonkey columns (including text, ntext, and image).
  - Included columns do not count against the 900 byte index key limit enforced by SQL Server.

Index Design Tasks

- Understand the characteristics of the database (OLTP vs OLAP)
- Understand the characteristics of the most frequently used queries
- Understand the characteristics of the columns used in the queries
- Determine the optimal storage location for the index.

General Index Design Guidelines

- Database considerations
  - Too many indexes on a table affect the performance of INSERT, UPDATE, DELETE, MERGE statements
  - Indexing small tables may not be optimal
  - Indexes on views are useful when views contain aggregations and/or table joins
- Query considerations
  - Create nonclustered indexes for columns frequently used in WHEREs and JOINs
  - Covering indexes can improve query performance
  - Write queries that insert or modify as many rows as possible in a single statement
  - Evaluate the query type and how columns are used in the query

General Index Design Guidelines

- Column Considerations
  - Keep the length of index key short for clustered indexes
  - Clustered indexes are better on unique/nonnull cols
  - Columns of **ntext**, **text**, **image**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)** cannot be specified as index key columns
  - Examine column uniqueness
  - Examine data distribution in column (avoid indexes on columns with few unique values) – use filtered indexes
  - Consider the order of the columns for mutliple index. Columns used in an equal to (=), greater than (>), less than (<), or BETWEEN search condition should be placed first. Additional columns should be ordered from the most distinct to the least distinct.
  - Consider indexing computed columns.

Unique Indexes

- A unique index guarantees that the index key contains no duplicate values
- Specifying a unique index makes sense only when key columns are unique
- Uniqueness – helpful information for query optimizer

Filtered Indexes

Filtered Index: an optimized nonclustered index, especially suited to cover queries that select from a well-defined subset of data

```
CREATE NONCLUSTERED INDEX FI_EndDate ON
Products (ProductID,  EndDate)
 WHERE EndDate IS NOT NULL ;
GO
```

- Improved query performance
- Reduced index maintenance costs
- Reduced index storage costs

Indexes for Deletes

At DELETE:

- SQL Server will check for dependent rows by examining all foreign keys
- It will then check any related tables for data.
    - If there is an index, SQL Server will use that index to check for related data
    - If there isn't an index, though, SQL Server will have to scan the table for data.
- Deletes could be very slow if there is no index defined for foreign keys

Indexed Views

SET options	Required value	Default server value
ANSI_NULLS	ON	ON
ANSI_PADDING	ON	ON
ANSI_WARNINGS	ON	ON
ARITHABORT	ON	ON
CONCAT_NULL_YIELDS_NULL	ON	ON
NUMERIC_ROUNDABORT	OFF	OFF
QUOTED_IDENTIFIER	ON	ON

Indexed Views Restrictions

- SELECT statement cannot reference other views
- All functions must be deterministic
- AVG, MIN, MAX, STDEV, STDEVP, VAR and VARP are not allowed
- The index must be both clustered and unique
- SELECT statement must not contain subqueries, outer joins, EXCEPT, INTERSECT, TOP, UNION, ORDER BY, DISTINCT etc

Columnstore Indexes

- groups and stores data for each column and then joins all the columns to complete the whole index
- Suited for warehouses (read only tables)

Hard and fast rules for indexing

- Each table should have a clustered index that is (ideally) small, selective, ever increasing, and static. (a table without a clustered index is called a *heap*.)
- Implement nonclustered indexes on foreign key relationships
- Implement nonclustered indexes on columns that are frequently used in WHERE clauses.
- Do not implement single-column indexes on every column in a table. This will cause high overhead.
- In multi-column indexes, list the most selective (nearest to unique) first in the column list.
- For most often-used queries create covering nonclustered index.

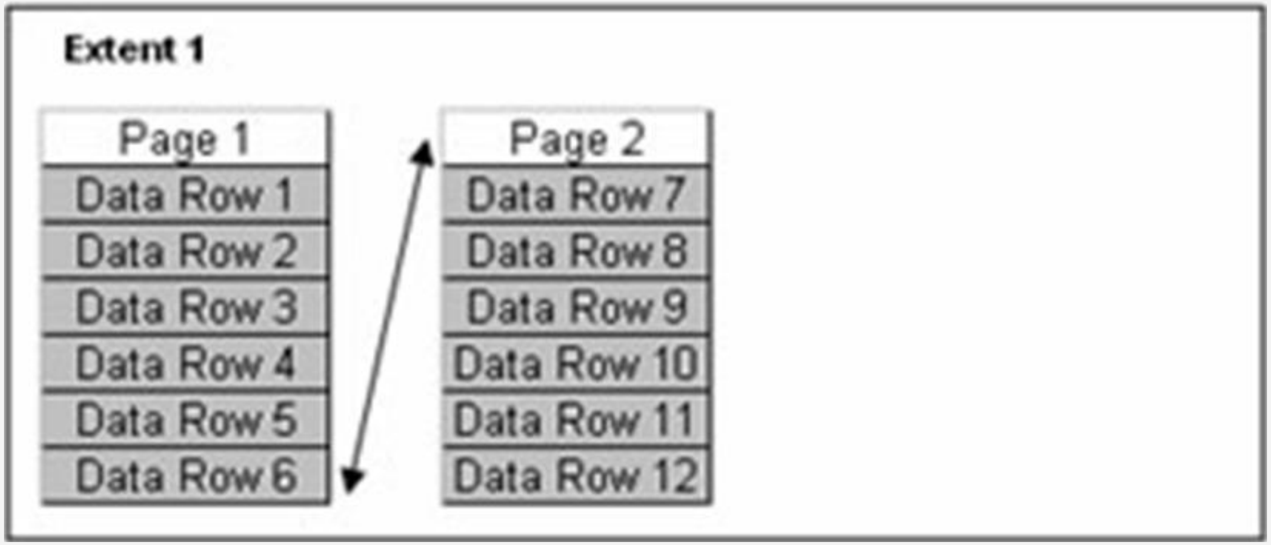
Fragmentation

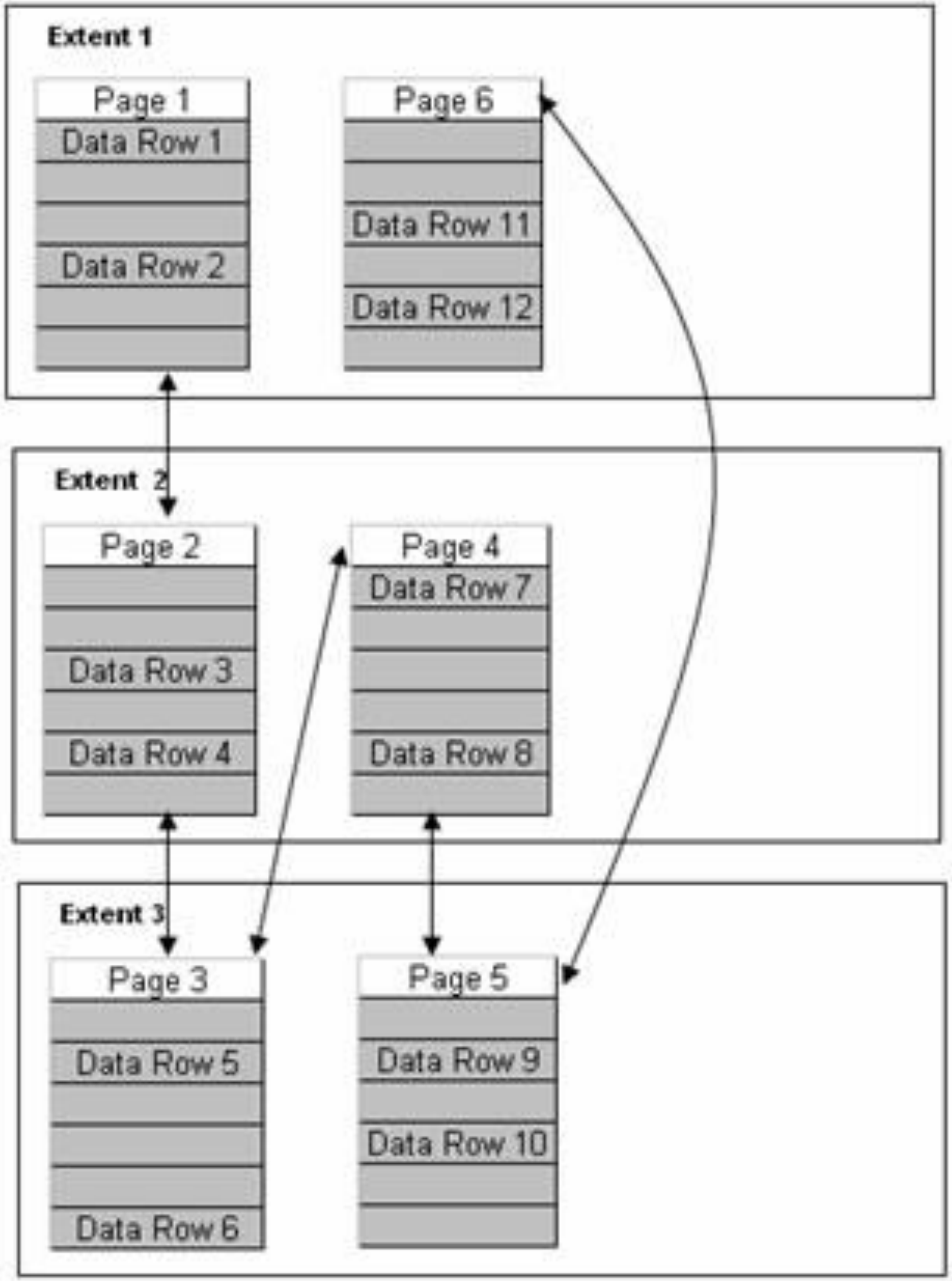
- *Internal Fragmentation*: records are stored non-contiguously inside the page. Internal fragmentation occurs if there is unused space between records in a page. The fullness of each page can vary over time. This unused space causes poor cache utilization and more I/O, which ultimately leads to poor query performance.
- *External Fragmentation*: When on disk, the physical storage of pages and extents is not contiguous. When the extents of a table are not physically stored contiguously on disk, switching from one extent to another causes higher disk rotations.

Fragmentation

- *Logical Fragmentation*: Every index page is linked with previous and next page in the logical order of column data. Because of Page Split, the pages turn into *out-of-order* pages.
- An *out-of-order* page is a page for which the next physical page allocated to the index is not the page pointed to by the next-pag*e* pointer in the current leaf page.

Page read requests: 2
Extent switches: 0
Disk space used by table: 16 KB
avg_fragmentation_in_percent: 0
avg_page_space_used_in_percent: 100





Page read requests: 6
Extent switches: 5
Disk space used by table: 48 KB
avg_fragmentation_in_percent > 80
avg_page_space_used_in_percent: 33

Fragmentation

- *sys.dm_db_index_physical_stats*
  - **avg_fragmentation_in_percent**: This is a percentage value that represents external fragmentation.
  - **avg_page_space_used_in_percent:** This is an average percentage use of pages that represents to internal fragmentation.

- **Reducing Fragmentation in a Heap:**
  - To reduce the fragmentation of a heap, create a clustered index on the table.
  - Creating the clustered index: rearrange the records in an order, and then place the pages contiguously on disk.

Fragmentation

Reducing Fragmentation in a Index:

- If avg_fragmentation_in_percent > 5% and < 30%, then use ALTER INDEX REORGANIZE:
  - reorder the leaf level pages of the index in a logical order.
- If avg_fragmentation_in_percent > 30%, then use ALTER INDEX REBUILD:
  - replacement for DBCC DBREINDEX to rebuild the index online or offline. In such case, we can also use the drop and re-create index method.
- Drop and re-create the clustered index:
  - Re-creating a clustered index redistributes the data and results in full data pages. The level of fullness can be configured by using the FILLFACTOR option in CREATE INDEX.