SQL Server: Why Physical Database Design Matters

Module 3: Data Types and Index Size

Kimberly L. Tripp
Kimberly@SQLskills.com
http://www.SQLskills.com/blogs/Kimberly

pluralsight
hardcore developer training

Introduction: Does Data Type Choice Impact Indexes?

- **Again, you might be thinking: disk space is cheap – who cares?…**
- **Once you understand the basic indexes structures, you'll see how profound the effect of your choice can be**
- **We'll discuss the impact of data type choice by index type:**
  - Clustered index
  - Nonclustered index
  - Columnstore index
- **Throughout this module, I'll be discussing/demonstrating:**
  - Key considerations around index structures
  - What the physical structures look like and how to analyze them
  - How does SQL Server access data (based on index usage)
  - What is the effect on performance

What Structures Exist for a Table?

- **Table structure is either:**
  - Unordered: the table is called a heap
  - Ordered: through creation of a clustered index and the table is called a clustered table (not to be confused with other RDBMS' clustered table)
- **Indexes**
  - Clustered: only one can exist per table as this defines the data's order
    - Not *required*, but <u>highly</u> recommended
    - The data is physically ordered at create/rebuild
    - The data is logically ordered through a doubly-linked list
    - Cluster key choice is CRITICAL!
  - Nonclustered: not required, can have up to 999 of these (249 in 2000/2005)
    - DO NOT affect the base table's structure
    - Are affected by whether or not the table is clustered
  - **Hint: The nonclustered index dependency on the clustered index should impact your choice for the clustering key!**

What About Columnstore Indexes?

- **Traditional clustered and nonclustered indexes are also known as "row-based" indexes**
- **Columnstore indexes are new in SQL Server 2012 and have a completely different internal structure as values for a single column are stored together**
  - Data type has an impact but data distribution is even more interesting (in terms of possible column-level compression)
- **These are beyond the scope of this course and have very specific/limited uses**
  - SQL Server 2012: max of one nonclustered columnstore index per table and once created, the table is read-only
  - SQL Server 2014: columnstore indexes can be clustered and read/write
- **Columnstore indexes will not be discussed here, check out Joe Sack's course *SQL Server 2012: Nonclustered Columnstore Indexes***

Table Structure: Heap

- A table without a clustered index
- Records are not ordered and there is no doubly-linked list
- Accessed via allocation structures only so if no indexes exist then a full table scan is required for any SELECT query
- Imagine 80,000 records at 20 rows/per page = 4,000 pages
- Table scan costs <u>at least</u> 4,000 I/Os
  - Why "at least"?

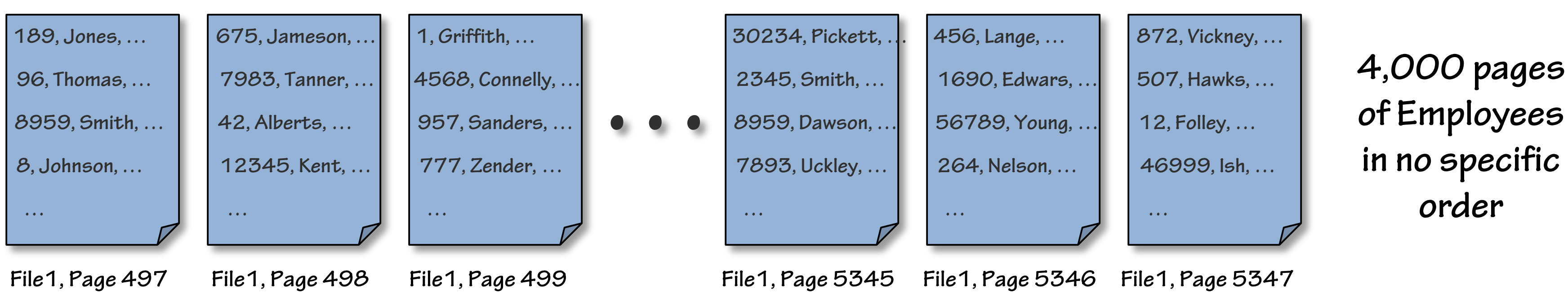
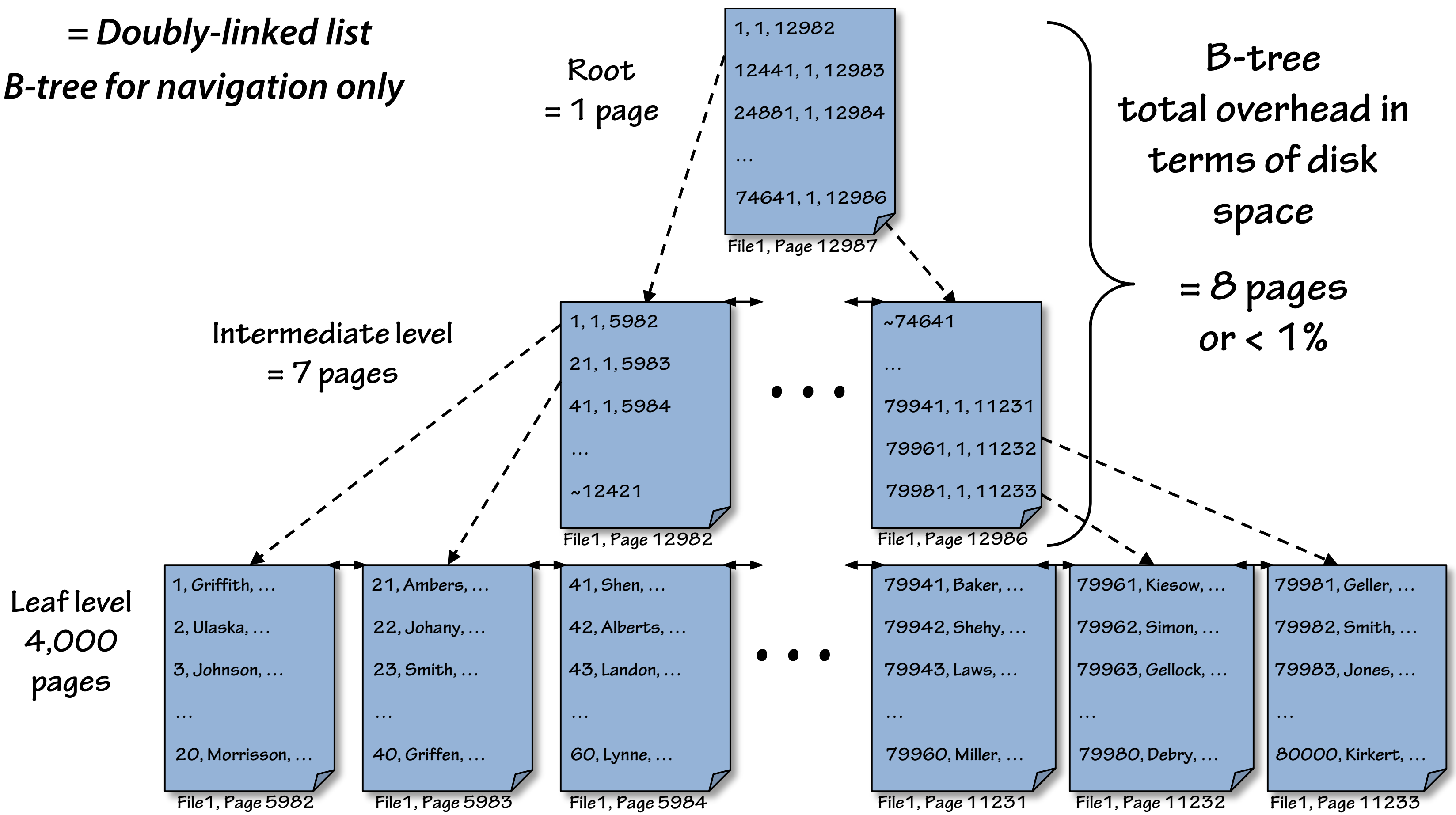


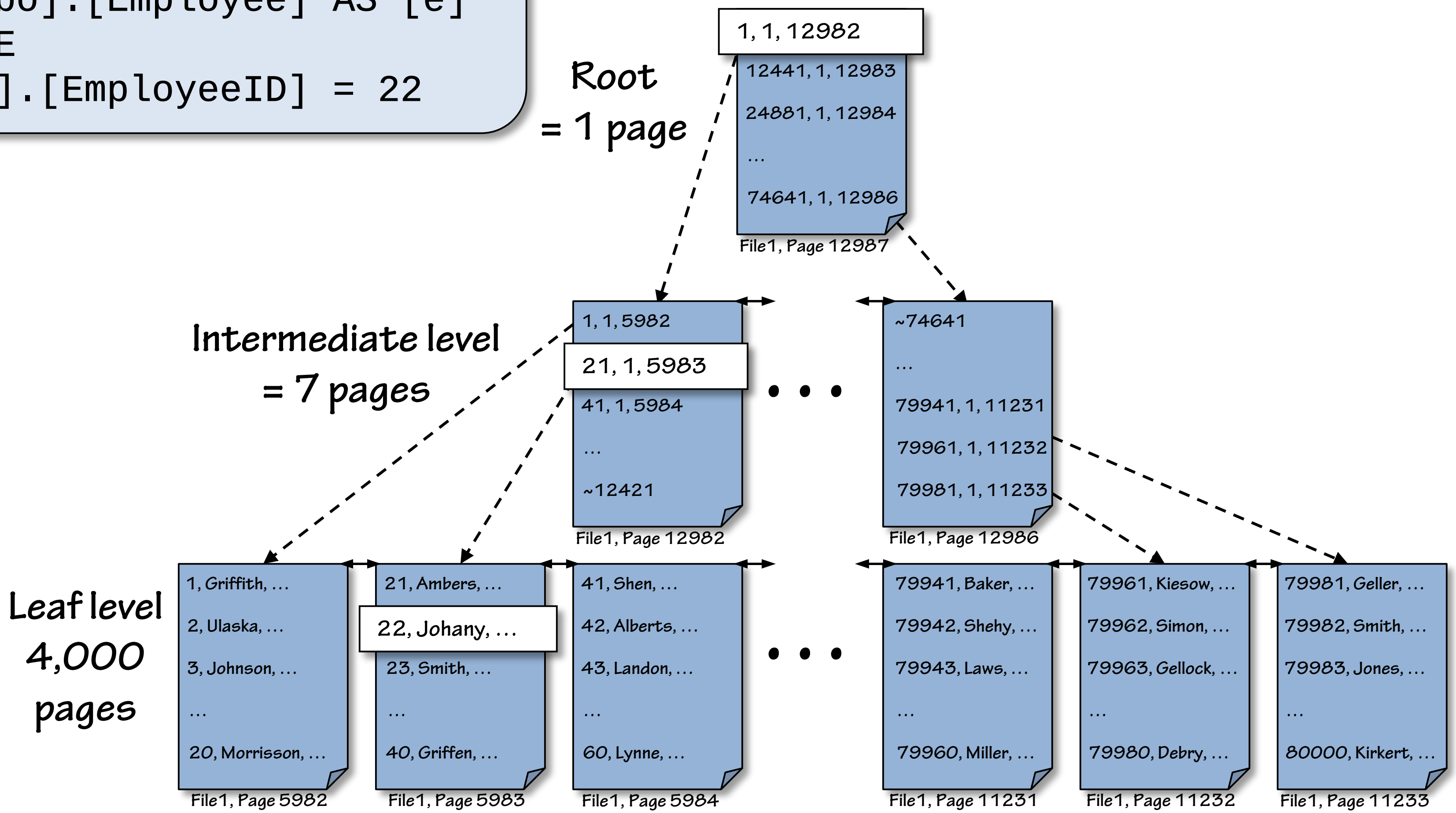
Table Structure: Clustered Table

Row Data
= Ordered leaf level
= Doubly-linked list
B-tree for navigation only



Accessing Data Using a Clustered Index

```
SELECT [e].*
FROM
  [dbo].[Employee] AS [e]
WHERE
  [e].[EmployeeID] = 22
```



Nonclustered Indexes: The Book Analogy

- Think of a book with indexes in the back
- The book has one form of logical ordering
- To lookup data, you use the indexes in the back
- Using a "Common Name" you look up that value in the index
- Once you find that index value then, you need to lookup the actual data based on its page in the book… i.e. a "bookmark" lookup
- The bookmark always depends on the book's content order

Index – Species Common Name

Index – Animal by Type, Name
Bird, Mammal, Reptile, etc…

Index – Animal by Country, Name

Index – Animals by Habitat, Name
Air, Land, Water

Index – Species Scientific Name

Index – Animal by Continent, Country, Name

Accessing Data Using a Nonclustered Index

- **Nonclustered indexes work a lot like an index in the back of a book:**
  - Nonclustered index leaf level is the data defined by the index, in sorted order (exactly the same as a book)
  - Nonclustered index b-tree doesn't exist in a book but does in SQL Server.
    - Its use is just like the b-tree in a clustered index, i.e. navigational only
- **Nonclustered indexes use the clustering key as the lookup value**
  - Instead of a page number (or, physical locator – which could change), SQL Server uses the clustering key as the lookup ID in a nonclustered index
    - Can be both good OR bad, depending on the key
- **Remember: the nonclustered index dependency on the clustered index should impact your choice for the clustering key!**
  - This is why…

Clustering Key Usage in Nonclustered Indexes

Imagine the internals of a nonclustered index on SocialSecurityNumber on three different versions of the Employee table each with a different clustering key

SSN	Lookup	Uniquifier
000-00-0184	Smith	0 (0 bytes)
000-00-0236	Jones	1 (4 bytes)
000-00-0395	Smith	1 (4 bytes)
000-00-0418	Jones	0 (0 bytes)

The lookup value is non-unique (and wide as the column type of nvarchar(40)).
Also, what if there are multiple rows with the same lastname (Smiths/Jones/Anderson)?

CL: Lastname

SSN	Lookup
000-00-0184	92CF41D7-17BF-49F7-B5C8-D3246C19B302
000-00-0236	2F87EEBB-FBA1-4C06-B7F1-BE63285B5935
000-00-0395	2EF09CA4-6E48-47AA-A688-3D9FDEA220E0
⋮	⋮

The lookup value is a GUID = 16 bytes

CL: GUID

SSN	Lookup
000-00-0184	31101
000-00-0236	22669
000-00-0395	18705
⋮	⋮

The lookup value is an int = 4 bytes

CL: EmployeeID

Each table starts at 80,000 rows over 4,000 pages (due to the average row size of 400 bytes/row and therefore 20 rows/page). Then EACH/EVERY index must include the (entire) lookup value.

Clustering Key Widens Nonclustered Indexes

- **Imagine a real-world scenario**
  - Table has 8 nonclustered indexes and 10 million rows
- **What's the overhead required (and total space) for the bookmark lookups in the nonclustered indexes:**
  - With a clustering key of an int (4 bytes)
  - With a clustering key of a GUID (16 bytes)
  - With a natural key (6 columns and ~64 bytes)
    - NOTE: This is just the overhead of the data type without factoring in nullable/non-unique.

Simple calculations for overhead in the LEAF level of the nonclustered indexes based on CL key columns defined		
CL Key Column(s)	Width of CL key (bytes)	MB
int	4	305.18
datetime	8	610.35
datetime, int	12	915.53
guid	16	1,220.70
composite	32	2,441.41
composite	64	4,882.81

Nonclustered Index Overhead

- Table has 8 nonclustered indexes and 10 million rows
- What is the required disk space for placing the clustering key in each and every nonclustered index
- Add required overhead for nullability as well as whether the column is unique vs. non-unique
- Keys of: int / bigint / datetime, int / GUID are likely to be unique and non-nullable (marked with *)

NOTE: Did not factor in additional overhead for composite keys and the number of variable-width columns they might have.

CL Key Column(s)	Bytes	MB
int *	4	305.18
int, nullable	7	534.06
int, non-unique (min)	4	305.18
int, non-unique (max)	12	915.53
int, non-unique (min), nullable	7	534.06
int, non-unique (max), nullable	15	1,144.41
bigint *	8	610.35
bigint, nullable	11	839.23
datetime, int *	12	915.53
datetime, int, nullable	15	1,144.41
guid *	16	1,220.70
guid, nullable	19	1,449.58
composite 32 bytes (comp32) *	32	2,441.41
comp32, nullable	35	2,670.29
comp32, non-unique (min)	32	2,441.41
comp32, non-unique (max)	40	3,051.76
comp32, non-unique (min), nullable	35	2,670.29
comp32, non-unique (max), nullable	43	3,280.64
composite 64 bytes (comp64) *	64	4,882.81
comp64, nullable	67	5,111.69
comp64, non-unique (min)	64	4,882.81
comp64, non-unique (max)	72	5,493.16
comp64, non-unique (min), nullable	67	5,111.69
comp64, non-unique (max), nullable	75	5,722.05
composite 128 bytes (comp128) *	128	9,765.63
comp128, nullable	131	9,994.51
comp128, non-unique (min)	128	9,765.63
comp128, non-unique (max)	136	10,375.98
comp128, non-unique (min), nullable	131	9,994.51
comp128, non-unique (max), nullable	139	10,604.86

Is it Really That Much Space?

- What about 100 million rows with 12 nonclustered indexes?

Simple calculations for overhead in the LEAF level of the nonclustered indexes based on CL key columns defined		
CL Key Column(s)	Width of CL key (bytes)	MB
int	4	4,577.64
bigint	8	9,155.27
datetime, int	12	13,732.91
guid	16	18,310.55
composite32, nullable	35	40,054.32
composite64, nullable	67	76,675.42
composite128, nullable	131	149,917.60

- You're looking at ***gigabytes*** of storage, memory, backups
- Insert/update performance (logging)
- Maintenance requirements
- My point is that it really does add up
- It is something you need to strategize/analyze and DESIGN!

Clustered Index Criteria

- **How do you keep your clustering key as streamlined as possible?**
  - Unique
    - Yes: No extra time/space overhead, data takes care of this criteria
    - NO: SQL Server must “uniquify” the rows on INSERT
  - Static
    - Yes: Reduces overhead
    - NO: Costly to maintain during updates to the key
  - Narrow
    - Yes: Keeps the nonclustered indexes narrow
    - NO: Unnecessarily wastes space
  - Non-nullable/fixed-width
    - Yes: Reduces overhead
    - NO: Adds overhead to ALL nonclustered indexes
  - Ever-increasing key value
    - Yes: Reduces index fragmentation
    - NO: Inserts/updates might cause significant index fragmentation

Choose a GOOD Clustering Key

- **Identity column**
  - Adding this column and clustering on it can be extremely beneficial, even when you don't "use" this data
- **DateCol, identity**
  - Composite key defined in that order
    - Do not use date alone as that would need to be "uniquified"
  - Great clustering key for partitioned tables
  - Ideal where you have a lot of data-related queries (even if not partitioned)
- **GUID**
  - NO: if populated by client-side call to .NET client to generate the GUID
    - OK as the primary key but not as the clustering key
  - NO: if populated by server-side NEWID() function
    - OK as the primary key but not as the clustering key
  - Maybe: if populated by the server-side NEWSEQUENTIALID() function as it creates a more sequential pattern (and therefore less fragmentation)
    - But, this isn't really why you chose to use a GUID…

Primary Key does NOT have to be the Clustering Key

- **Primary key: relational integrity**
- **Clustering key: internal mechanism for looking up rows (bookmark lookup)**
- **SQL Server enforces uniqueness of a primary key through an index and defaults to clustered**
  - 1 clustered index per table
  - 1 primary key per table
- **If the primary key is a natural key then you probably want to enforce it with a nonclustered index**
  - Might be very wide
  - Very expensive to duplicate in each and every nonclustered index
- **If the table doesn't have a column (or small set of columns) that meets these criteria then consider adding a surrogate [identity] key and then create the clustered index on it**

Scenario: What is the Real Cost?

- **AdventureWorksDW.dbo.FactInternetSales**
- **Clustered index (composite index of two seemingly narrow columns):**
  - SalesOrderNumber type: nvarchar(20)
  - SalesOrderLineNumber type: tinyint
- **Nonclustered indexes (all, single-column nonclustered):**
  - IX_FactIneternetSales_ShipDateKey: ShipDateKey
  - IX_FactInternetSales_CurrencyKey: CurrencyKey
  - IX_FactInternetSales_CustomerKey: CustomerKey
  - IX_FactInternetSales_DueDateKey: DueDateKey
  - IX_FactInternetSales_OrderDateKey: OrderDateKey
  - IX_FactInternetSales_ProductKey: ProductKey
  - IX_FactInternetSales_PromotionKey: PromotionKey
- **Everything seems narrow and somewhat optimal, until we talk about the data types of the clustering key columns**

Scenario: What's in That Key?

- **What? What does the data look like?**
  - SalesOrderNumber = 7 characters (SO12345) which is 14 bytes of data
    - EVERY row is SO + 5-digit number – why?
    - Then, add variable-width overhead (each column has 2 bytes in the variable block as an offset)
    - When this is the first (or only) variable-width column then the addition of a variable block within the row adds 2 more bytes
  - Each of these 7 character "numbers" requires 18 bytes (14 + 2 + 2)
- **If the clustering key requires it then EVERY nonclustered index requires it**
- **7 nonclustered indexes:**
  - Ironically, ALL columns (of all nonclustered indexes) are:
    - Non-nullable and fixed-width
  - They do not require a variable-block on their own, but when you add the clustering key

Scenario: What's the Total Cost?

- **What's the physical cost of this poorly defined column: SalesOrderNumber vs. an int (and ditching the type since there's only one type in all of the data)**
- **14 bytes wasted per row, per index**
- **7 nonclustered indexes x 14 = 98 bytes (completely wasted) per row**
- **What if the table were larger?**
  - Imagine 10 million rows and 10 nonclustered indexes:
    - 10,000,000 x 140 / 1024 / 1024 = **1.335 GB** of nonsense
  - Imagine 100 million rows and 10 nonclustered indexes:
    - 100,000,000 x 140 / 1024 / 1024 = **13.35 GB** of nonsense
  - Imagine 1 billion rows and 12 nonclustered indexes:
    - 1,000,000,000 x 154 / 1024 / 1024 = **143.42 GB** of nonsense

Summary: The Effect of Data Type Choice

- **Data type choice can have a profound affect on:**
  - Column size
  - Row size
  - Index size
- ***Everything* is less efficient/effective when design is poor**
  - Waste disk space
  - Waste cache (and this is still very costly)
  - Larger backups, more time to backup
  - Logging is more costly with wider rows (DML is negatively affected)
  - Maintenance is more costly (and possibly required more)
  - Queries can be less efficient
    - Required to put data into memory than really needed
    - Returning more data than what's necessary