SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Module 6: Index Internals

Kimberly L. Tripp

Kimberly@SQLskills.com

Overview

- **Index concepts**
- **Table structure**
- **Index internals**
  - Heaps
  - Why cluster
  - Table usage
  - Employee table case study
- **Clustering key columns in nonclustered indexes**
- **Indexing for Performance**
  - What do we know?
  - What should we do?
  - Suggestions for the clustering key!

Index Concepts: Tree Analogy

- If a tree were data and you were looking for leaves with a certain property, you would have two options to find that data….
- 1) Touch every leaf, interrogating each one to determine if they held that property…SCAN
- 2) If those leaves (which had that property) were grouped such that you could start at the root, move to the branch and then directly to those leaves…SEEK

Nonclustered Indexes: Book Analogy

- Think of a book with indexes in the back
- The book has one form of logical ordering
- For references you use the indexes in the back… to find the data in which you are interested you look up the key
- When you find the key you must lookup the data based on its location… i.e. a "bookmark" lookup
- The bookmark always depends on the (book) content order

Index – Species Common Name

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

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

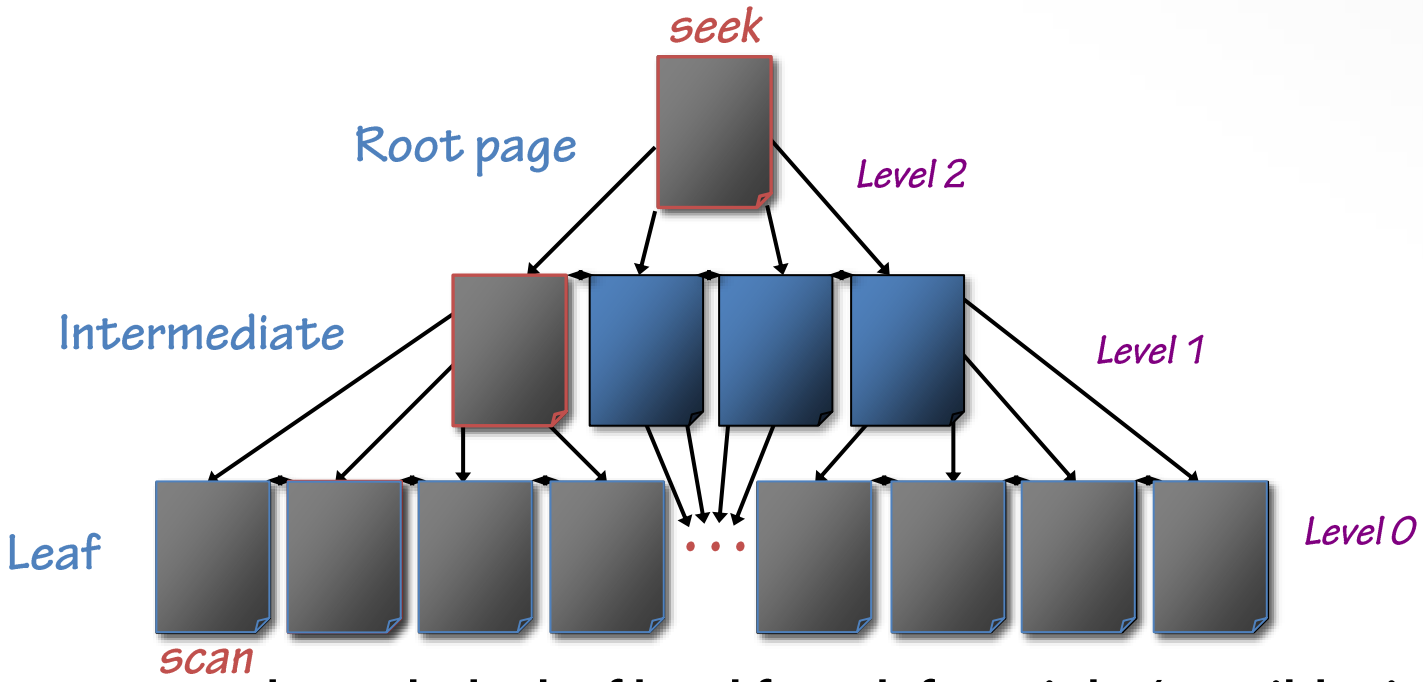
Index – Species Scientific Name

Index – Animal by Country, Name

Index – Animal by Continent, Country, Name

Seek vs. Scan

- Seek: starts at the root and uses the tree structure to move from top to bottom



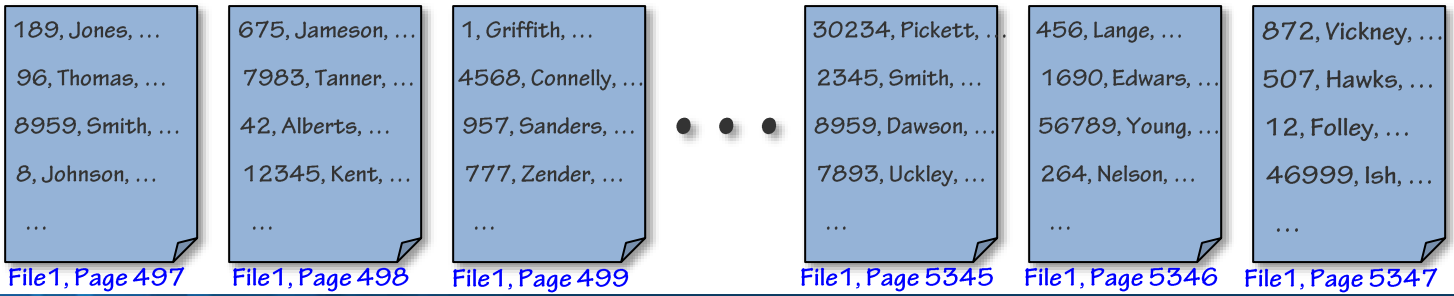
- Scan: moves through the leaf level from left to right (possibly right to left)

Table Structure Overview

- Heap: a table without a clustered index
- Clustered table: a table with a clustered index
- Nonclustered indexes DO NOT affect the base table's structure
- However, nonclustered indexes 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!

Table Structure: Heap

- Heap: a table without a clustered index
- Records are NOT ORDERED, no doubly-linked list
- Access via Index Allocation Map (IAM)
  - IAMs = 8KB page (chain) which tracks object usage
    - 1 IAM chain per PARTITION (heap or b-tree)
    - For each partition, 1 IAM page per file, per 4GB for each allocation unit (data (in-row), LOB, row-overflow)
- If NO indexes exist then a full table scan is required
- 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"?)



4,000 pages of Employees in no specific order

Heap: Pros

- **Excellent for data loading**
  - Create empty table
  - Use multiple source files to take advantage of parallel data loading
  - Parallel index creation, after data load
- **Scenario**
  - 800,000 rows in single text.csv file
  - Empty HEAP, load data, build CL index, then 2 nonclustered indexes (21.800 sec with 0 to minimal fragmentation)
  - Empty clustered table with 2 nonclustered indexes, load data (64.223 sec with lots of fragmentation)
    300% SLOWER

Heap: Pros

- Effective for "staging" data
- Excellent for loading into a partition for a partitioned table or partitioned view
- Indexes can be created after load
- Efficient for SCANs ONLY, when no UPDATEs (otherwise, forwarding pointers so scans become significantly less efficient)
- Space efficient as freed space from deletes is re-used on subsequent inserts (at the cost of performance)

- See whitepaper: The Data Loading Performance Guide
  - http://bit.ly/1D5AOYS

Heap: Cons

- **Insert performance compromised**
  - Reclaims space v. perform
  - Lookup in IAM/PFS expensive if table has DELETEs and INSERTs

See KB Article Q297861
"Poor Performance on a HEAP"

- **Scenario**
  - 800,000 rows originally, delete % 17 = 47,058 gaps
  - HEAP with NO NC indexes – INSERT 50,000 rows (50.463 sec)
  - Clustered table with NO NC indexes – INSERT 50,000 rows (43.168 sec)

    15% Faster

Heap: Cons

- **Fixed 8-byte RID assigned on INSERT**
  - 2 for fileID, 4 for pageID, 2 for slot number (which defines the record offset on the page)
- **Rows can have forwarding pointers**
  - If modification results in record relocation
- **Forwarding pointers**
  - Benefit in nonclustered index RID Lookup (NC to data)
  - Negative for table scans (negative for OLAP/DSS)
    - Table with 0 forwarding pointers
    - TABLE SCAN I/Os = # of pages in table
    - Table with n forwarding pointers
    - TABLE SCAN I/Os = # of pages in table + n forwarding pointers

Heap: Cons

- **Scenario (802,942 Rows)**
  - Alter table – add new VARCHAR column
  - Update 14% of the data (both require Table Scans)
  - Clustered table
    - Update (19.456 sec) **35% Faster**
    - Size 75,976KB **170% Larger**
  - Heap
    - Update (29.903 sec)
    - Size 44,680KB

So the clustered table is faster but seems to waste space (but with proper maintenance…)
BUT is that the only difference?

Heap Issues: Why Cluster?

- **Heap has overhead**
  - Space
    - Fixed RID assigned on INSERT
    - Forwarding pointers (from record relocation)
  - Time
    - Negative for table scans (negative for OLAP/DSS)
  - Optimized for saving space (on INSERT)
  - Optimized for data loading – when empty and no updates (parallel data loading!!!)
- **Clustered tables**
  - Usually a better choice for OLTP or mixed workloads
  - Require consistent/automated maintenance

What Do We Know?

- Heaps offer excellent benefits for staging tables
- For OLTP/DS tables, user based modifications (not batch), performance is better with a clustered index
- However, CL indexes require administrative maintenance to alleviate negatives with regard to space
- Are all clustered indexes going to give the same gains?
- For true performance gains you must have the ***RIGHT*** clustered index!

Clustered Index Overview

- Not required, although highly recommended
- Only one per table
- Physical order applied at creation
- Logical order maintained through a doubly-linked list
- Requires ongoing and automated maintenance
- Need to choose wisely!

Clustered Index Key Choice
Historically

- **Clustering key chosen to remove hot spots… *Why?***
  - Page-level locking
- **Clustering key chosen to improve "range" query performance… *Why?***
  - Low selectivity "ranges" are obviously not bad, but are they the best?
- **Dependency on the clustered index was greater… *Why?***
  - SQL Server ONLY used ONE index per table per query
  - Adding nonclustered indexes or making nonclustered indexes wider degraded performance – without adding significant benefits
  - Pre-7.0 SQL Server used a VOLATILE RID for lookup (requiring significant index maintenance on DML – therefore fewer NC were desired)

Clustered Index Key Choice
Currently

- **Clustering key choice DOES NOT need to remove hot spots… *Why?***
  - True row-level locking
- **Clustering key choice is NOT the best for "range" queries… *Why?***
  - The CL key only gives ONE "range" query better performance – and only for queries asking for SELECT *
  - Range queries can be answered by better nonclustered indexes
  - SQL Server has improved index capabilities as indexes can be joined, scanned with lookups, aggregates, …
- **Dependency on the clustered index has CHANGED… *Why?***
  - Nonclustered indexes INCLUDE the clustering key for lookup
    - Unique: each row must be uniquely referenced from NC to CL (should not allow NULL)
    - Narrow: the CL key value is stored within EVERY NC Index
    - Static: if the value changes, ALL NC indexes need the change

Lookups

How is the Clustering Key Used in Nonclustered Indexes?

Imagine the internals of a nonclustered index on SocialSecurityNumber with 3 different versions of the Employee table with different clustering keys

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 an nvarchar(40)), what if there are two (or more?) Smith / 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.

Nonclustered Index Structures

- **A nonclustered index row has a minimum of:**
  - Header (TagA bytes) = 1 byte (fewer requirements than a data row structure)
  - Index columns (What data types? Are they all fixed length?)
- **A nonclustered index row might have:**
  - Null bitmap (minimum of 3 bytes)
    - Might be in the tree when the nonclustered index has NULLable columns*
    - Only *required* in the leaf level when the index has NULLable columns*
  - Variable block, but only if there are columns that are variable length (this requires 2 bytes for every variable-length column plus a 2-byte count of variable-length columns)
  - NOTE: A uniquifier is considered variable-length so this adds at least the 4-byte integer. If this is the only variable-length column, 4 additional bytes are needed for the variable-length offset (2 bytes) + counter (2 bytes)
- ***In SQL Server 2012, nonclustered indexes will have always have:**
  - A null bitmap in the leaf level of the index
    - Minimum of 3 bytes -> 1 bit per column (and a 2-byte column count)
  - Prior to 2012, a null bitmap only exists if there are NULLable columns in the index
  - Upgraded databases won't add the null bitmap until an index rebuild

The Impact of the Clustering Key on the Nonclustered Index Structures

- **Unique clustered index on an int**
  - B-tree key size: 11 bytes
    - 1 byte TagA, 4 bytes (int), 6 bytes (page pointer)
- **Non-unique clustered index on an int**
  - Minimum B-tree key size: 11 bytes (same as above)
  - Maximum B-tree key size: 19 bytes
    - 1 byte TagA, 4 bytes (int), 6 bytes (page pointer), 4 bytes for uniquifier, 2 bytes for variable offset, 2 bytes for counter
- **Non-unique, nullable clustered index on an int**
  - Minimum B-tree key size: 14 bytes
    - 1 byte TagA, 4 bytes (int), 6 bytes (page pointer), 2 bytes for null bitmap and 1 byte for actual null values (table only has 3 columns)
  - Maximum B-tree key size: 22 bytes
    - Same as above but add on the 4 bytes for uniquifier, 2 bytes for variable offset, 2 bytes for counter

hidden slide w/extra details

Lookups: What is the Impact?

How Does the Clustering Key Impact Nonclustered Indexes?

- Each nonclustered must "include" the entire clustering key either explicitly (in the nonclustered index definition) or implicitly (SQL Server adds the columns that are not already present)
- The wider the clustering key, the wider (and probably unnecessarily wider) your nonclustered indexes
- What about modifications?
- Does this really have that much of an impact??
  - Imagine the prior examples on a 10 million row table with 8 NC indexes

Simple calculations for overhead in the LEAF level of the nonclustered indexes based on CL key				
Description	Width of CL key	Rows	NC Indexes	MB
Unique clustered index on an int	4	10,000,000	8	305.18
Non-unique clustered index on an int (minimum)	4	10,000,000	8	305.18
Non-unique clustered index on an int (maximum)	12	10,000,000	8	915.53
Non-unique, nullable clustered index on an int (minimum)	7	10,000,000	8	534.06
Non-unique, nullable clustered index on an int (maximum)	15	10,000,000	8	1,144.41

hidden slide
w/extra details

Lookups

Nonclustered Indexes are Wider!

- **Imagine these costs in a real world scenario…**
  - 10 million rows, 8 nonclustered indexes
- **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 an GUID (16 bytes)
  - With a really wide clustering 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				
Description	Width of CL key	Rows	NC Indexes	MB
int	4	10,000,000	8	305.18
datetime	8	10,000,000	8	610.35
datetime, int	12	10,000,000	8	915.53
guid	16	10,000,000	8	1,220.70
composite	32	10,000,000	8	2,441.41
composite	64	10,000,000	8	4,882.81

Lookups
NC Leaf Overhead

- **Factor in whether or not the key is unique or not**
- **Generally, int / bigint / datetime, int / datetime, bigint OR GUID are likely to be unique**
- **Composite keys**

  Did not factor overhead for variable-width columns (minimum for variable block is 2 (for counter) + 2 for EACH variable column's offset into the variable block

Description	Bytes	Rows	NC Indexes	MB
int *	7	10,000,000	8	534.06
int, non-unique (min)	7	10,000,000	8	534.06
int, non-unique (max)	15	10,000,000	8	1,144.41
bigint *	11	10,000,000	8	839.23
bigint, non-unique (min)	11	10,000,000	8	839.23
bigint, non-unique (max)	19	10,000,000	8	1,449.58
datetime, int *	**15**	**10,000,000**	**8**	**1,144.41**
datetime, bigint *	**19**	**10,000,000**	**8**	**1,449.58**
guid *	19	10,000,000	8	1,449.58
composite 32 bytes (comp32) *	35	10,000,000	8	2,670.29
comp32, non-unique (min)	35	10,000,000	8	2,670.29
comp32, non-unique (max)	43	10,000,000	8	3,280.64
				0.00
composite 64 bytes (comp64) *	67	10,000,000	8	5,111.69
comp64, non-unique (min)	67	10,000,000	8	5,111.69
comp64, non-unique (max)	75	10,000,000	8	5,722.05
				0.00
composite 128 bytes (comp128) *	131	10,000,000	8	9,994.51
comp128, non-unique (min)	131	10,000,000	8	9,994.51
comp128, non-unique (max)	142	10,000,000	8	10,833.74

* Unique

PRIOR to 2008R2

Lookups

NC Leaf Overhead

- Factor in nullability and non-unique…
- int / bigint /datetime, int /GUID are likely to be unique
- Composite keys
  - Did not factor number of variable-width columns (minimum for variable block is 2 (for counter) + 2 for EACH variable column's offset into the variable block

* Unique & non-nullable

Description	Bytes	Rows	NC Indexes	MB
int *	4	10,000,000	8	305.18
int, nullable	7	10,000,000	8	534.06
int, non-unique (min)	4	10,000,000	8	305.18
int, non-unique (max)	12	10,000,000	8	915.53
int, non-unique (min), nullable	7	10,000,000	8	534.06
int, non-unique (max), nullable	15	10,000,000	8	1,144.41
bigint *	8	10,000,000	8	610.35
bigint, nullable	11	10,000,000	8	839.23
datetime, int *	12	10,000,000	8	915.53
datetime, int, nullable	15	10,000,000	8	1,144.41
guid *	16	10,000,000	8	1,220.70
guid, nullable	19	10,000,000	8	1,449.58
composite 32 bytes (comp32) *	32	10,000,000	8	2,441.41
comp32, nullable	35	10,000,000	8	2,670.29
comp32, non-unique (min)	32	10,000,000	8	2,441.41
comp32, non-unique (max)	40	10,000,000	8	3,051.76
comp32, non-unique (min), nullable	35	10,000,000	8	2,670.29
comp32, non-unique (max), nullable	43	10,000,000	8	3,280.64
composite 64 bytes (comp64) *	64	10,000,000	8	4,882.81
comp64, nullable	67	10,000,000	8	5,111.69
comp64, non-unique (min)	64	10,000,000	8	4,882.81
comp64, non-unique (max)	72	10,000,000	8	5,493.16
comp64, non-unique (min), nullable	67	10,000,000	8	5,111.69
comp64, non-unique (max), nullable	75	10,000,000	8	5,722.05
composite 128 bytes (comp128) *	128	10,000,000	8	9,765.63
comp128, nullable	131	10,000,000	8	9,994.51
comp128, non-unique (min)	128	10,000,000	8	9,765.63
comp128, non-unique (max)	136	10,000,000	8	10,375.98
comp128, non-unique (min), nullable	131	10,000,000	8	9,994.51
comp128, non-unique (max), nullable	139	10,000,000	8	10,604.86

hidden slide w/extra details

Lookups
Nonclustered Indexes are Wider!

- Or, what about 100 million rows w/12 nonclustered indexes

Simple disk space calculations of *JUST* the CL costs in the NC leaf level!				
Description	**Width of CL key**	**Rows**	**NC Indexes**	**MB**
int	7	100,000,000	12	8,010.86
bigint	11	100,000,000	12	12,588.50
datetime, int	15	100,000,000	12	17,166.14
datetime, bigint	19	100,000,000	12	21,743.77
guid	19	100,000,000	12	21,743.77
composite32, nullable	35	100,000,000	12	40,054.32
composite64, nullable	67	100,000,000	12	76,675.42

- You're looking at GBs of storage, memory, backups and fundamentally even insert/update performance as well as maintenance requirements.
- My point – it ***really*** does add up! It IS something you want to CHOOSE and DESIGN!

Scenario: What is the Real Cost?

AdventureWorksDW: FactInternetSales

- **Clustered index:**
  - SalesOrderNumber
  - SalesOrderLineNumber

Data type:
`nvarchar(20)`
`tinyint`

- **Nonclustered indexes:**
  - 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

Demo

AdventureWorks

The impact of key choice on nonclustered indexes

Scenario: What is the Real Cost?

AdventureWorksDW: FactInternetSales

- **Clustered index:**
  - Nonclustered leaf row REQUIRES variable block
    - SalesOrderLineNumber = 7 characters on average (SO12345) which is 14 bytes + variable (2 bytes in the variable block MINIMUM) but if the variable block is required then 2 more bytes for counter… 18 bytes
- **Nonclustered indexes:**
  - 7 nonclustered indexes – ALL columns (of every index) are non-nullable and fixed width so…
    - 14 bytes wasted per row, per index
    - 7 * 14 = 98 bytes (completely wasted) per row…
    - Imagine 10 million rows and 10 nonclustered indexes:
      - 10000000 * 140 / 1024 / 1024 = ***1.335GB*** of nonsense
    - Imagine 100 million rows and 10 nonclustered indexes:
      - 100000000 * 140 / 1024 / 1024 = ***13.35GB*** of nonsense
    - Imagine 1 billion rows and 12 nonclustered indexes:
      - select 1,000,000,000 * 154 / 1024 / 1024 = ***143.42GB*** of nonsense

Clustered Index Criteria

Keeping our 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 NC indexes narrow
  - NO: Unnecessarily wastes space
- Non-nullable/fixed-width
  - Yes: Reduces overhead
  - NO: Adds overhead to ALL nonclustered indexes
- Ever-increasing
  - Yes: Reduces fragmentation
  - NO: Inserts/updates might cause splits (significant fragmentation)

Clustering on an Identity

- **Naturally unique**
  - Should be combined with constraint to enforce uniqueness
- **Naturally static**
  - Should be enforced through permissions and/or trigger
- **Naturally narrow**
  - Only numeric values possible, whole numbers with scale = 0
- **Naturally non-nullable/fixed-width**
  - An identity column cannot allow nulls, a numeric is fixed-width
- **Naturally ever-increasing**
  - Creates a beneficial hot spot…
  - Needed pages for INSERT already in cache
  - Minimizes cache requirements
  - Helps reduce fragmentation due to INSERTs
  - Helps improve availability by naturally needing less defrag

Clustering Key Suggestions

- **Identity column**
  - Adding this column and clustering on it can be extremely beneficial – even when you don't "use" this data
- **DateCol, bigint (identity?)**
  - In that order and as a composite key (not date alone as that would need to be "uniquified")
  - Great for partitioned tables
  - Great for ever increasing tables where you have a lot of date-related queries
- **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…
- **Key points: unique, static, as narrow as possible, and less prone to require maintenance – by design**

Clustering on an Identity
The Bad

- **Problem: can create system page contention on allocation when there are lots of tables that each have high insert volume**
  - Each table makes its allocation request from the GAM
  - Can create contention (especially true in high object creation environments: eg. Tempdb)
- **Solution:**
  - More effective RAID arrays
  - Multiple files in the filegroup (for the large/critical table)
  - Isolating the object to its own filegroup
    - Potentially this will create a second problem
- **Basically, step 1 is that you want to make system allocation as fast as possible!**

Clustering on an Identity
The Bad

- **Problem: can create page latch contention on insert allocation in extremely high insert volume (usually 500+ per sec, per table)**
- **Solution: In 2019 CREATE INDEX WITH OPTIMIZE_FOR_SEQUENTIAL_KEY**
- **Solution: may want to consider SOME distribution of the insert workload**
  - **Good:** Create a composite clustering key to create multiple insertion points
    - Country, ID: If you do business in 6 countries then there will be 6 insertion points – better distributing the INSERT "hot spot"
    - Negative: you'll still have some fragmentation; you won't be getting range-based locality for scans
  - **Better:** Create multiple insertion points and have each set directed to their own file/filegroup
    - Example 1: Partition by country and place each country's data on a separate file/filegroup
      - Negative: more management overhead and potentially sizing complexity
    - Example 2: Hash-based partitioning using a persisted computed column using a simple modulo (and, can also support un-aligned indexes for nonclustered performance)
      - Negative: more management overhead (removes the sizing complexity and offers load balancing)
- **Basically, step 2 is that you want to make page/row allocation as fast as possible!**
- **FYI: PAGELATCH_EX waits and heavy inserts: http://tinyurl.com/o4xpxxb**

Clustering on an Identity
The Bad

- **What about "range" queries and optimization?**
- **Problem: tuning query performance then focuses on nonclustered indexes and indexed views**
- **Not really a problem: tuning low-selectivity/range queries with nonclustered indexes is NOT really a bad thing:**
  - Faster access to low selectivity range queries
  - Nonclustered indexes are used in numerous non-obvious ways (multiple nonclustered indexes can be joined to cover a query)
  - More flexible in definition (i.e. indexed view can include computations, substrings, etc. BUT [first index] must be CLUSTERED UNIQUE: if view already has a unique key definition it simplifies the indexed view)
  - Fragmented nonclustered indexes are easier to rebuild (only requires a shared table lock)
  - Nonclustered indexes are easier to keep less fragmented – i.e. more frequent rebuilds and fillfactor helps more because row is narrower/smaller

Key Constraints Create Indexes

- **Primary key constraint**
  - Defaults to unique clustered
  - Only one per table

```
ALTER TABLE Employee
  ADD CONSTRAINT EmployeePK
    PRIMARY KEY CLUSTERED (EmployeeID)
```

- **Unique key constraints**
  - Default to unique nonclustered
  - Maximum of 249 per table and up to 999 per table in SQL Server 2008+

```
ALTER TABLE Employee
  ADD CONSTRAINT EmployeeSSNUK
    UNIQUE NONCLUSTERED (SSN)
```

Primary Key Does NOT Have to Be the Clustering Key

- Primary key: relational integrity
- Clustering key: internal mechanism for looking up rows *(infamous bookmark lookup)*

- SQL Server enforces uniqueness of a primary key through an index and defaults to clustered
  - (1 CL index per table, 1 PK per table)
- If the primary key is a natural key then you probably want to enforce it with a 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 cluster it!

Nonclustered Index Overview

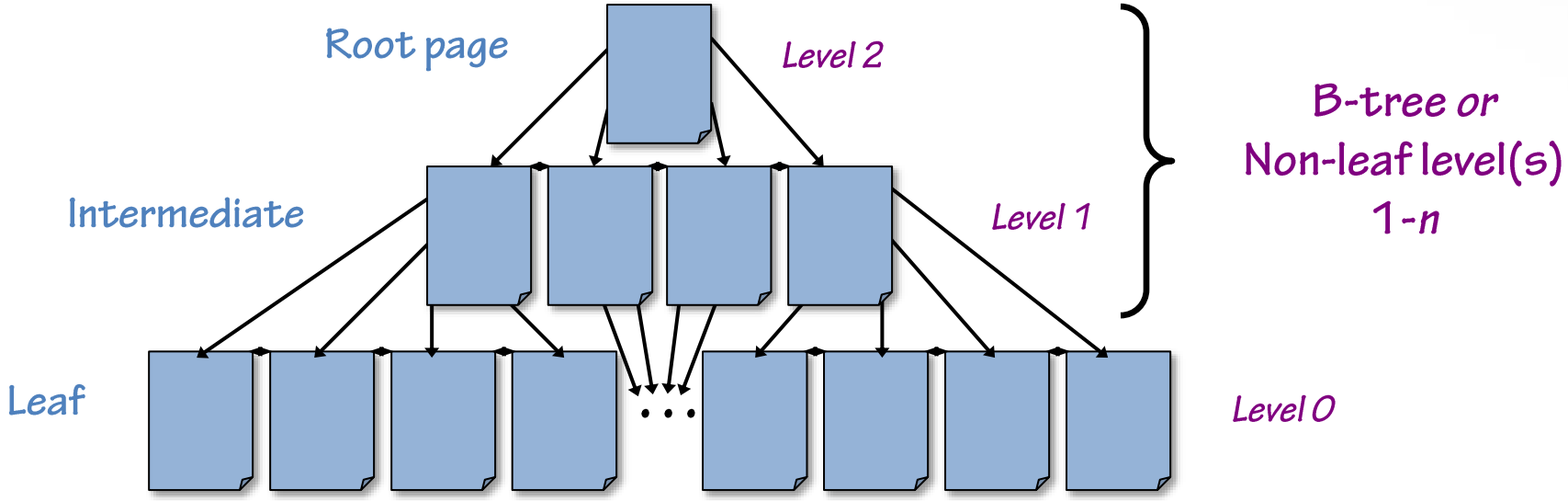
- Not required, although critical to achieving optimal performance
- Maximum of 249 per table and increased to 999 per table in SQL Server 2008+
- Leaf structure separate from base table
- Based on the heap's fixed RID or clustering key
- Logical order of index entries maintained through a doubly-linked list
- By far the 

type of index for range queries if it covers the query!

*Don't ask for *, limit your queries!!!*

Physical Index Levels
Generic Overview

- **Leaf level:** contains something for every row of the table in indexed order
- **Non-leaf level(s) or B-tree:** contains something, specifically representing the FIRST value, from every page of the level below. Always at least one non-leaf level. If only one, then it's the root and only one page. Intermediate levels are not a certainty.



Employee Table Case Study

- **Employee table assessments**
- **Clustered Employee table**
  - Physically order data
  - Add the B-tree (B+ tree, which means it's not kept balanced)
  - Complete math
  - Complete clustered index structure
- **Nonclustered unique constraint for SSN**
  - Build separate leaf level
  - Add the B-tree (B+ tree)
  - Complete math
  - Complete nonclustered index structure

Employee Table: Assessments

- Average row size = 400 bytes/row

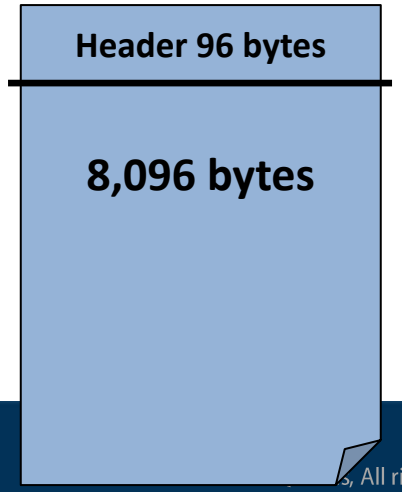
```
CREATE TABLE Employee
(
EmployeeID              Int             NOT NULL        Identity,
LastName                nvarchar(30)    NOT NULL,
FirstName               nvarchar(29)    NOT NULL,
MiddleInitial           nchar(1)        NULL,
SSN                     char(11)        NOT NULL,
…other columns…)
```

- 80,000 current Employees ∴ rows

$$\frac{\text{8,096 bytes / page}}{\text{400 bytes / row}} = \textbf{20 rows/page}$$

$$\frac{\text{80,000 employees}}{\text{20 rows / page}} = \textbf{4,000 pages}$$

8KB = 8,192 bytes



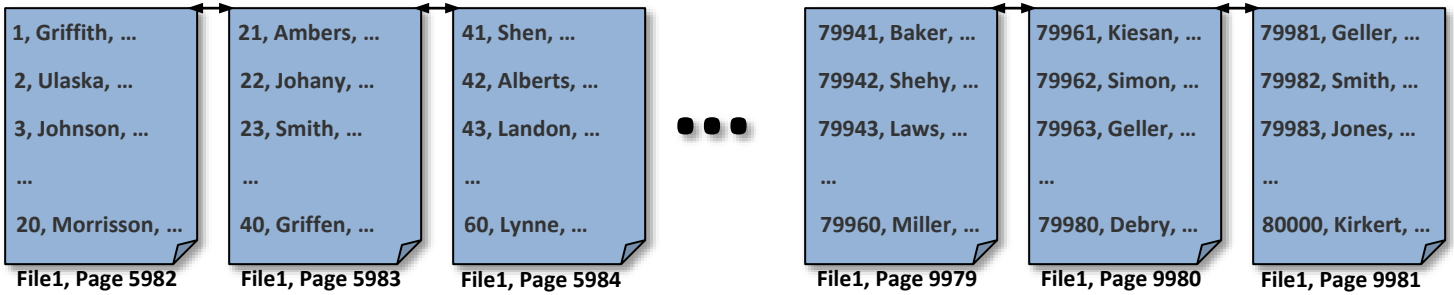
Clustered Employee Table

- Step 1: Physically order data

Review the index level definitions…
Does this seems to match one of the definitions?

Yes!
When a table is clustered
the data becomes the leaf level of the clustered index!

4,000 pages of Employees in clustering key order



Clustered Employee Table

Step 1: Physically order data

Step 2: Add the tree structure

starting from the leaf level and going up to a root of 1 page

B-tree entry = index key value + pointer + row overhead*

Pointer = page pointer of 6 bytes = 2 for fileID + 4 for pageID

Row overhead varies based on many factors

(min of 1 byte in the row)

Non-leaf level entry for clustered index on EmployeeID = 11

4 bytes for EmployeeID (int) + 6 bytes for page pointer
+ 1 byte for row overhead

$$\frac{\text{8,096 bytes / page}}{\text{11 bytes / entry + 2 bytes in slot array}} = \textbf{622 index entries per non-leaf level page}$$

How many entries to store? **4,000**

Remember – a non-leaf level contains one entry for every PAGE of the level below.

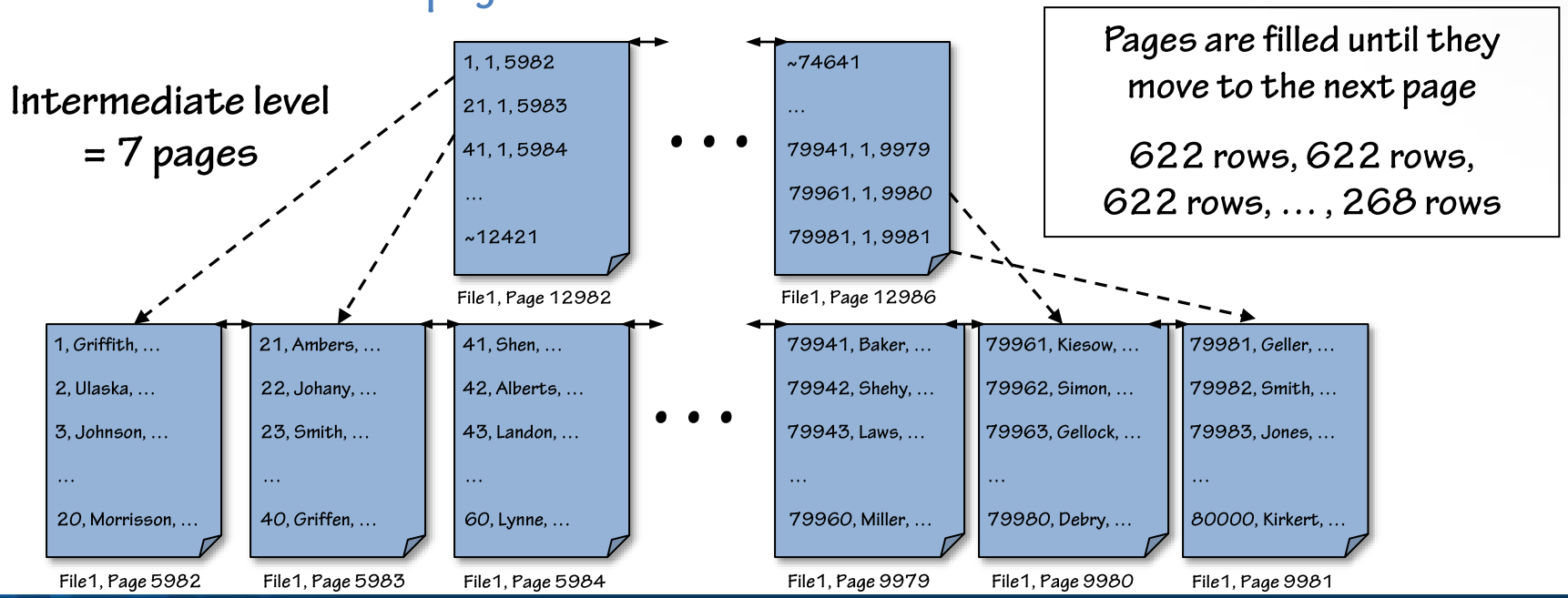
Clustered Employee Table

Step 1: Physically order data

Step 2: Add the B-tree

starting at the leaf level and working up to a root of 1 page

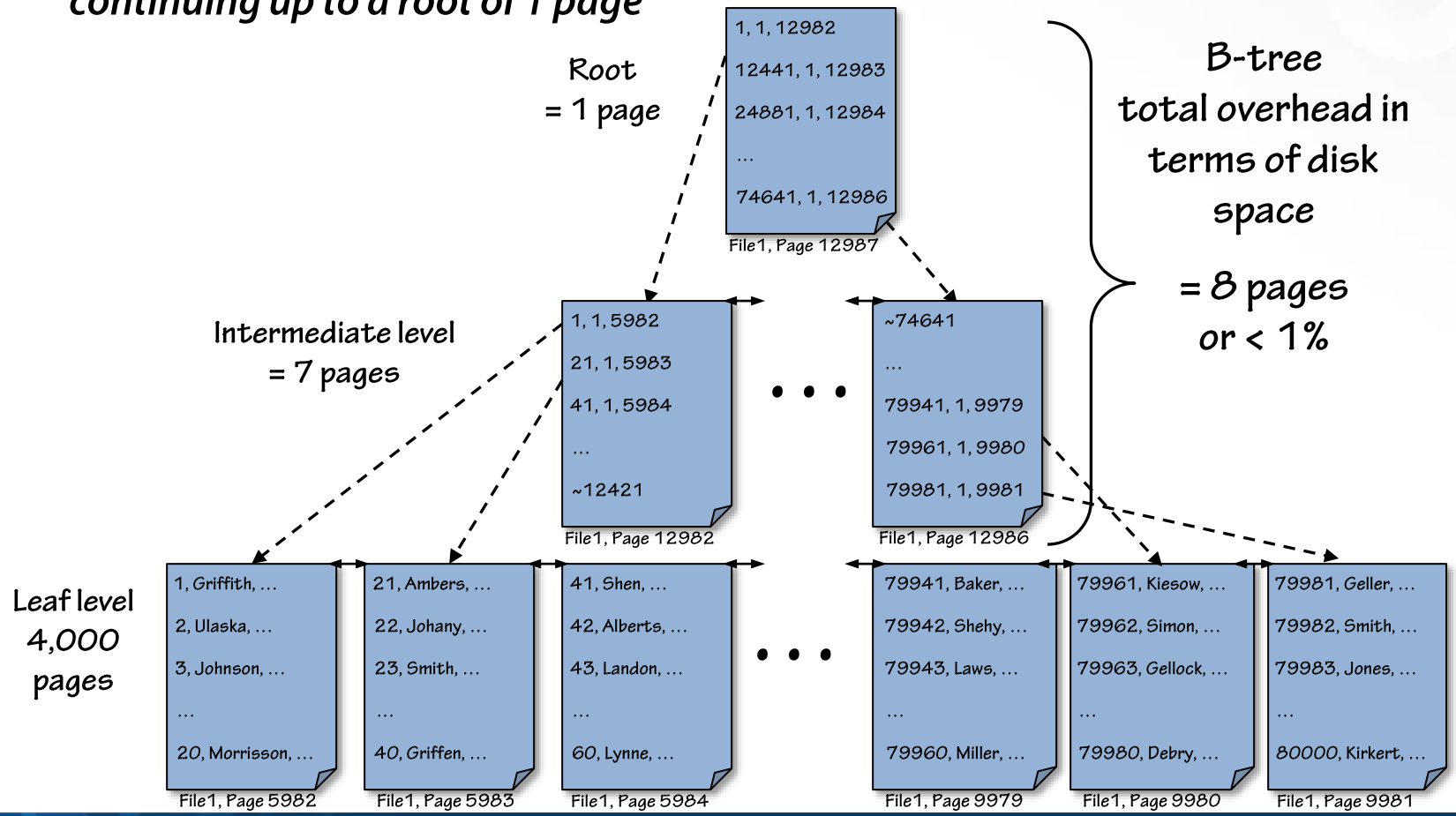
$$\frac{\text{4,000 entries to store}}{\text{622 entries/page}} = \text{7 pages in the first B-tree level}$$



Clustered Employee Table

Step 2: Complete the B-tree

continuing up to a root of 1 page



Nonclustered Index
Unique Constraint on SSN

- **Leaf level entry for nonclustered index**
  = NC index column(s) + row lookup ID + row overhead
  - Row lookup ID = fixed RID (if heap) or clustering key
  - Row overhead = TagA byte (1) + Null block (min of 3) + additional overhead as needed (are there variable-width columns?)

  = TagA (1 byte) + SSN (11 bytes) + EmployeeID (4 bytes) + Null block (3 bytes)
  = 19 bytes/entry + 2 bytes in the slot array
- **Entries per leaf level page**

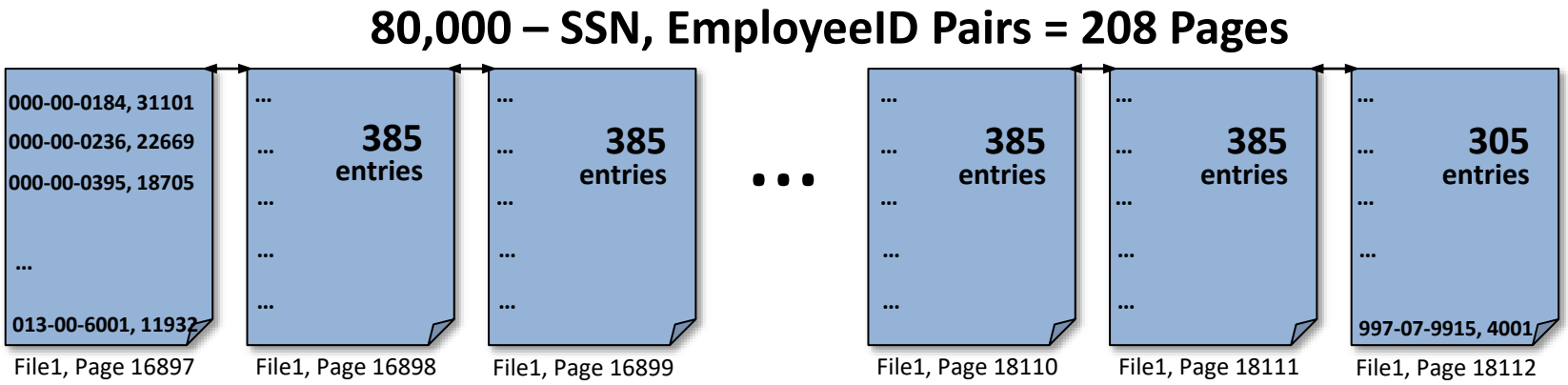
$$\frac{\text{8,096 bytes/page}}{\text{19 bytes/entry + 2 bytes in slot array}} = \textbf{385 index entries per leaf level page}$$

- **Pages for leaf level**

$$\frac{\text{80,000 rows}}{\text{385 rows/page}} = \textbf{208 pages}$$

Nonclustered Index
Unique Constraint on SSN

- The leaf level of the nonclustered index is built first…
- SQL Server will duplicate the SSN and EmployeeID for EVERY ROW and order it by the index definition (ascending by default).
- Every INSERT/DELETE will need to touch each nonclustered index; SQL Server will keep them up-to-date and current.



Nonclustered Index
Unique Constraint on SSN

- **Non-leaf level entry for nonclustered index**
  = NC index column(s) + row lookup ID* + pointer + row overhead
  - *The Row lookup ID is *only* included when the nonclustered is nonunique = fixed RID (if heap) or clustering key
  - Row overhead = TagA byte (1) + Null block (min of 3) but only when there are nullable columns in the index row + additional overhead as needed (are there variable-width columns?)

  = TagA (1 byte) + SSN (11 bytes) + pointer (6 bytes)
  = 18 bytes/entry + 2 bytes in the slot array

$$\frac{\text{8,096 bytes/page}}{\text{18 bytes/entry + 2 bytes in slot array}} = \textbf{404 index entries per non-leaf level page}$$

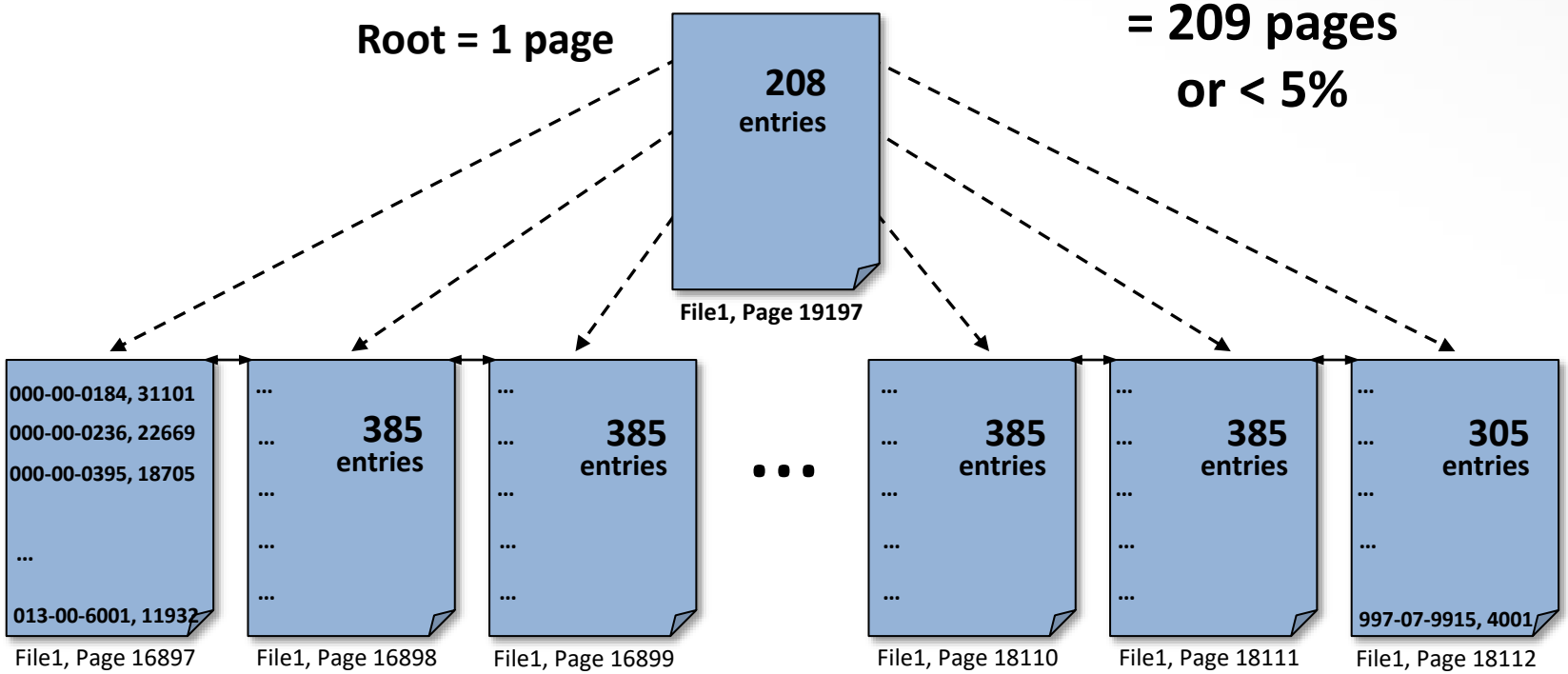
$$\frac{\text{208 rows}}{\text{404 rows/page}} = \textbf{1 page} \quad \textbf{= root}$$

Nonclustered Index

Unique Constraint on SSN

Total overhead in terms of disk space

= 209 pages or < 5%



Clustering Key Columns WHERE? (1 of 2)
Where do They Go Within Nonclustered Indexes?

- **What if:**

```
CREATE UNIQUE CLUSTERED INDEX IXCL
     ON tname (c6, c8, c2)
CREATE NONCLUSTERED INDEX IXNC1
     ON tname (c5, c2, c4)
```

  - Leaf level: c5, c2, c4, c6, c8
  - KEY/btree: c5, c2, c4, c6, c8

```
CREATE UNIQUE NONCLUSTERED INDEX IXNC1
     ON tname (c5, c2, c4)
```

  - Leaf level: c5, c2, c4, c6, c8
  - KEY/btree: c5, c2, c4

- **Key points:**
  - Clustering key columns are added only ONCE to your nonclustered indexes
  - Where they are added (leaf only or all the way up the tree) is based on whether or not the nonclustered is nonunique. When nonunique, the CL key goes up the tree.

Clustering Key Columns WHERE? (2 of 2)

Where do They Go Within Nonclustered Indexes?

Same clustered index:

```
CREATE UNIQUE CLUSTERED INDEX IXCL
  ON tname (c6, c8, c2)
```

If you NEED CL Key columns for seeking…

```
CREATE NONCLUSTERED INDEX IXNC1
    ON tname (c5, c2, c4)
```

- Leaf level: c5, c2, c4, c6, c8
- KEY/btree: c5, c2, c4, c6, c8
- Can seek on any left-based subset of the tree:
  - c5
  - c5, **c2**
  - c5, c2, c4
  - c5, c2, c4, c6
  - c5, c2, c4, c6, c8

```
CREATE UNIQUE NONCLUSTERED INDEX
IXNC1 ON tname (c5, c2, c4)
```

- Leaf level: c5, c2, c4, c6, c8
- KEY/btree: c5, c2, c4
- Can seek on any left-based subset of the tree:
  - c5
  - c5, c2
  - C5, c2, c4

```
CREATE NONCLUSTERED INDEX IXNC1
    ON tname (c5, c4)
```

- Leaf level: c5, c4, c6, c8, c2
- KEY/btree: c5, c4, c6, c8, c2
- Can seek on any left-based subset of the tree:
  - c5
  - c5, **c4**
  - c5, c4, c6
  - c5, c4, c6, c8
  - c5, c4, c6, c8, c2

```
CREATE UNIQUE NONCLUSTERED INDEX
  IXNC1 ON tname (c5, c4)
```

- Leaf level: c5, c4, c6, c8, c2
- KEY/btree: c5, c4
- Can seek on any left-based subset of the tree:
  - c5
  - c5, c4

reminder slide

Index Internals

What Do We Know?

- Clustered index leaf level IS the data
- Nonclustered index leaf level is duplicate data, in a separate structure and automatically maintained as changes occur
- B-trees are built on top of the leaf level up to a root of one page
- Nonclustered index is based on the clustered Index when the table is clustered…

Why do we need to know?

Index Internals
What Should We Do?

- **OLTP tables or mixed workload tables**
  - Consider a clustered index with an ever-increasing identity column
    - Creates a hot spot of activity, ensuring minimal cache requirements
    - Inserts won't cause splits
    - The clustering key is already unique
- **DSS/analysis tables**
  - Will want more nonclustered indexes so you still need to be aware of the clustering key size…
- **Characteristics of most/general importance:**
  - Narrow, unique, and static
  - Ever-increasing (reduced insertion points)

Review

- **Index concepts**
- **Table structure**
- **Index internals**
  - Heaps
  - Why cluster
  - Table usage
  - Employee table case study
- **Clustering key columns in nonclustered indexes**
- **Indexing for Performance**
  - What do we know?
  - What should we do?
  - Suggestions for the clustering key!

Questions!