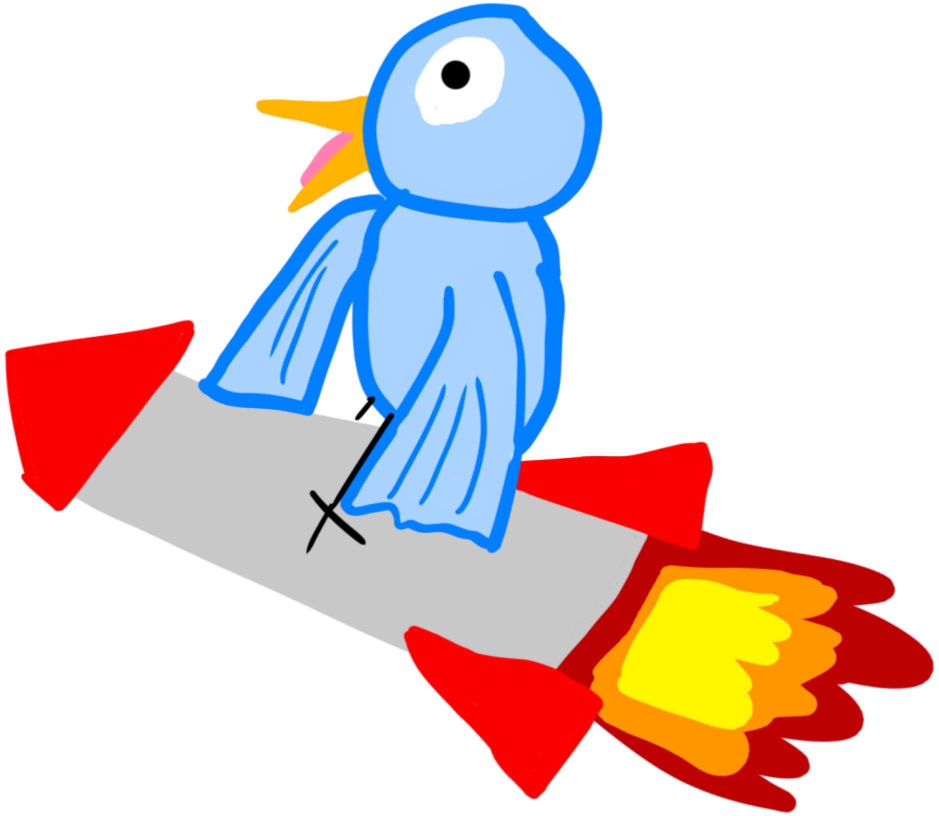
# I Want To Go Faster!

A Beginner's Guide to Indexing



@bertwagner bertwagner.com youtube.com/bertwagner bert@bertwagner.com

# Bert Wagner

SQL WITH BERT

YouTube

Slides available here!

@bertwagner bertwagner.com youtube.com/bertwagner bert@bertwagner.com

# Why Indexes?

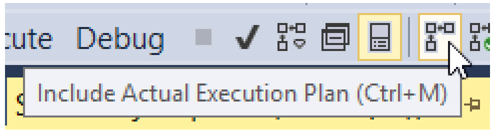
- Biggest bang for the buck
  - Can potentially fix many queries at once
- Positive downstream side effects
  - Can help:
    - Reduce blocking
    - Prevent deadlocking
    - Improve caching

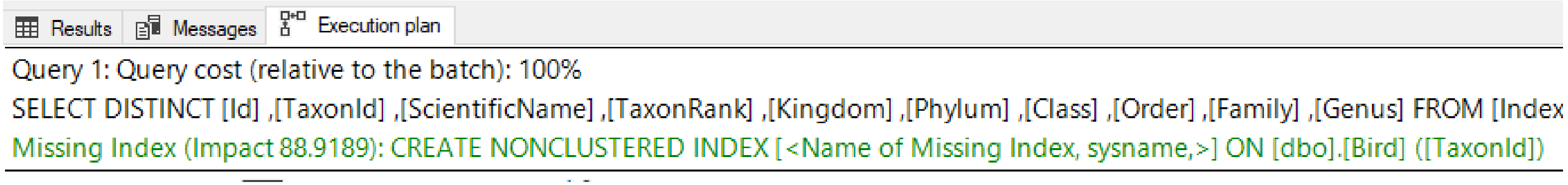
# Why NOT Indexes?

- They take up space
- Every index adds overhead on insert, updates, deletes
- Maintenance
  - Fragmentation – external and internal
  - Ownership
  - Maintenance windows introduce blocking, downtime, coordination

# When you need to index

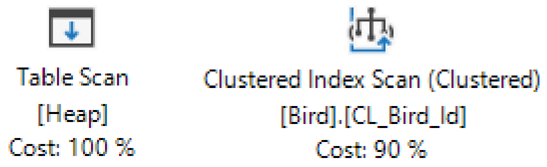
- You have many different queries running slowly against  a table (or multiple tables)
- You have one query with lots of executions is slow (refactor first)
- You see Microsoft’s green index recommendations in execution plans





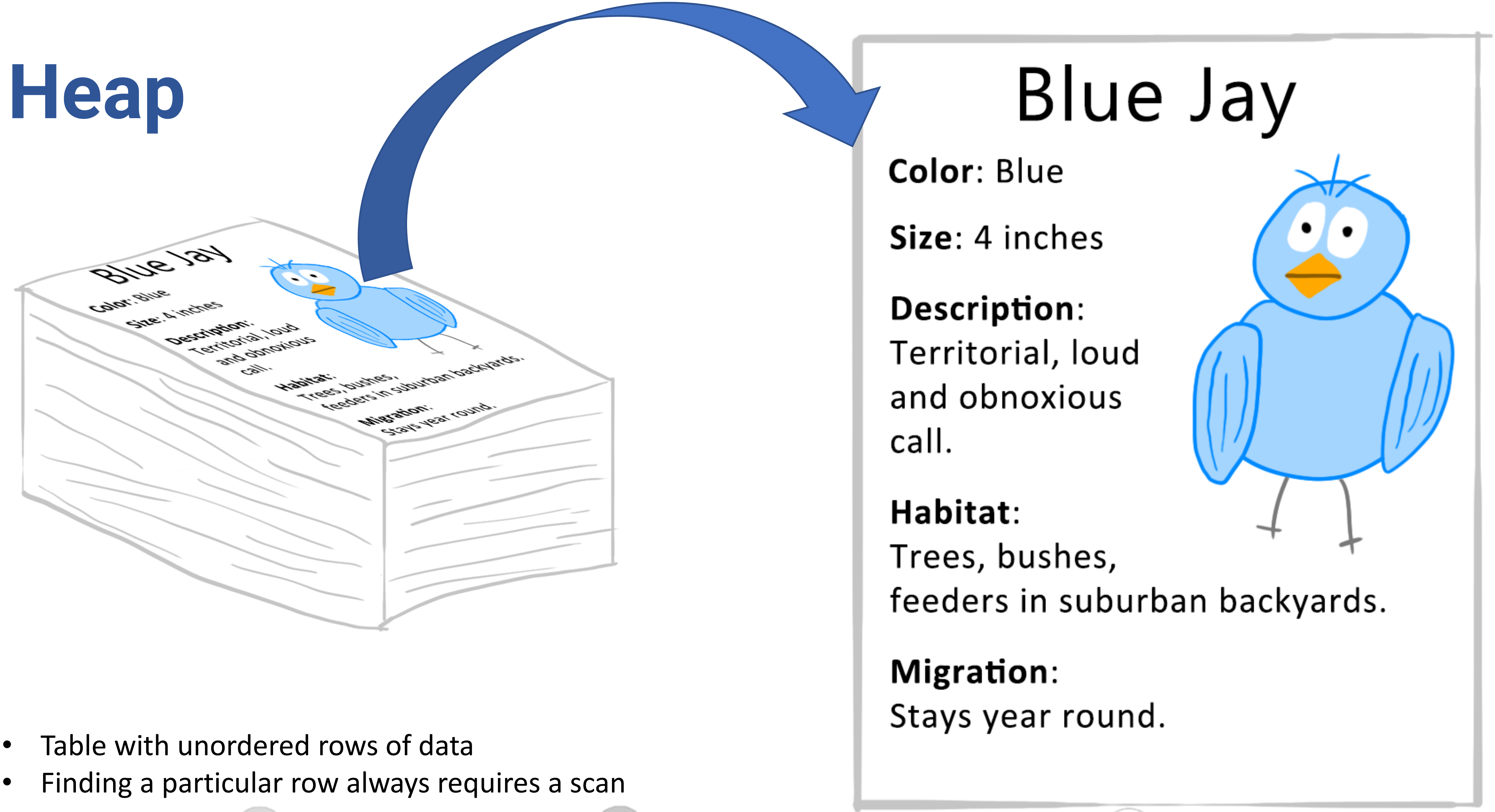
Query 1: Query cost (relative to the batch): 100%
SELECT DISTINCT [Id] ,[TaxonId] ,[ScientificName] ,[TaxonRank] ,[Kingdom] ,[Phylum] ,[Class] ,[Order] ,[Family] ,[Genus] FROM [Index
Missing Index (Impact 88.9189): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Bird] ([TaxonId])

- Seeing lots of scans in execution plans (not always bad)



# Index Internals

# Heap



## Blue Jay

**Color**: Blue

**Size**: 4 inches

**Description**:
Territorial, loud and obnoxious call.

**Habitat**:
Trees, bushes,
feeders in suburban backyards.

**Migration**:
Stays year round.

- Table with unordered rows of data
- Finding a particular row always requires a scan

# Heap

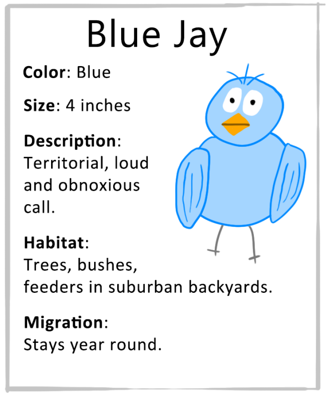
Blue Jay
Color: Blue
Size: 4 inches
Description:
Territorial, loud and obnoxious call.
Habitat:
Trees, bushes, feeders in suburban backyards.
Migration:
Stays year round.

```
CREATE TABLE dbo.FieldGuide
(
    BirdName nvarchar(100),
    Color nvarchar(20),
    Size tinyint,
    Description nvarchar(1000),
    Habitat nvarchar(1000),
    Migration nvarchar(1000)
)
```

# Clustered Index

Ordered Book of Birds
CL_BirdName

Ordered alphabetically by bird name.



Blue Jay

**Color**: Blue

**Size**: 4 inches

**Description**:
Territorial, loud and obnoxious call.

**Habitat**:
Trees, bushes, feeders in suburban backyards.

**Migration**:
Stays year round.

Cardinal

**Color**: Red

**Size**: 6 inches

**Description**:
Red with a mohawk on its head.

**Habitat**:
Trees, bushes, feeders in suburban backyards.

**Migration**:
Stays year round.

Sparrow

**Color**: Brown

**Size**: 2 inches

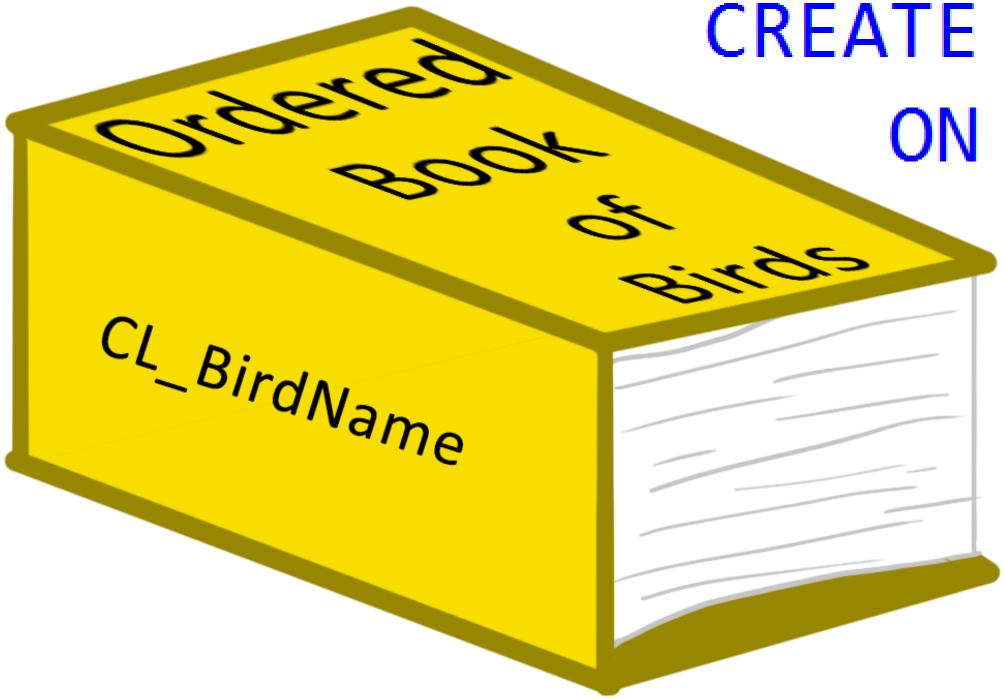
**Description**:
Little brown bird.

**Habitat**:
Tall bushes, thick growth.

**Migration**:
Stays year round.

- Default table order is defined – rows stored sorted in that order.
- Finding a particular row can now sometimes use a seek.

# Clustered Index



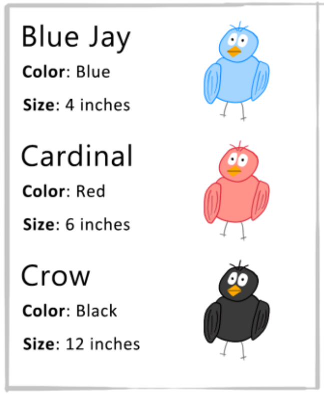
```
CREATE CLUSTERED INDEX CL_BirdName
       ON dbo.FieldGuide (BirdName);
```

Our index “key” – what order the data is stored in.

# Nonclustered Index

Small
Ordered
Book
Of
Birds
IX_BirdName
_Includes

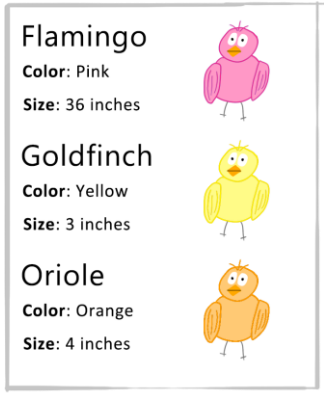
Ordered alphabetically by bird name… but with fewer columns



Blue Jay
**Color**: Blue
**Size**: 4 inches

Cardinal
**Color**: Red
**Size**: 6 inches

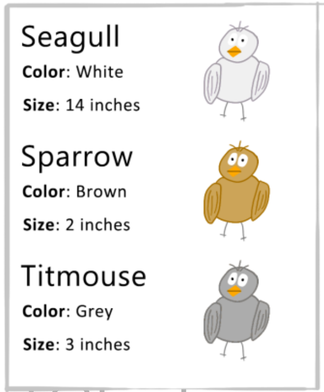
Crow
**Color**: Black
**Size**: 12 inches



Flamingo
**Color**: Pink
**Size**: 36 inches

Goldfinch
**Color**: Yellow
**Size**: 3 inches

Oriole
**Color**: Orange
**Size**: 4 inches



Seagull
**Color**: White
**Size**: 14 inches

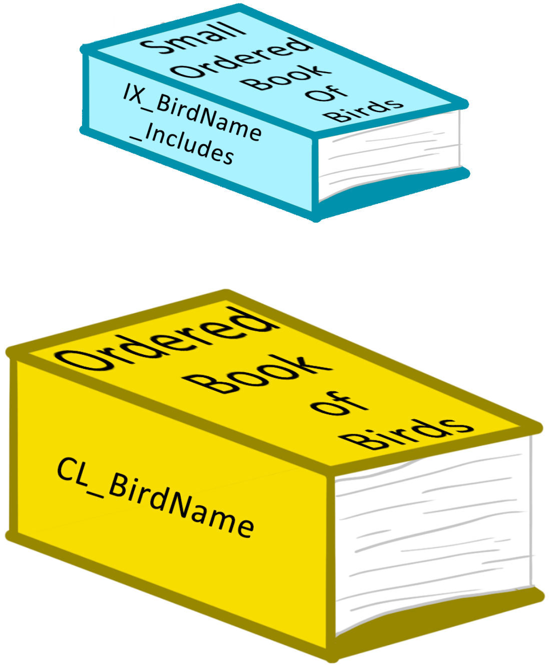
Sparrow
**Color**: Brown
**Size**: 2 inches

Titmouse
**Color**: Grey
**Size**: 3 inches

- Subset of columns
- Stored in a different order

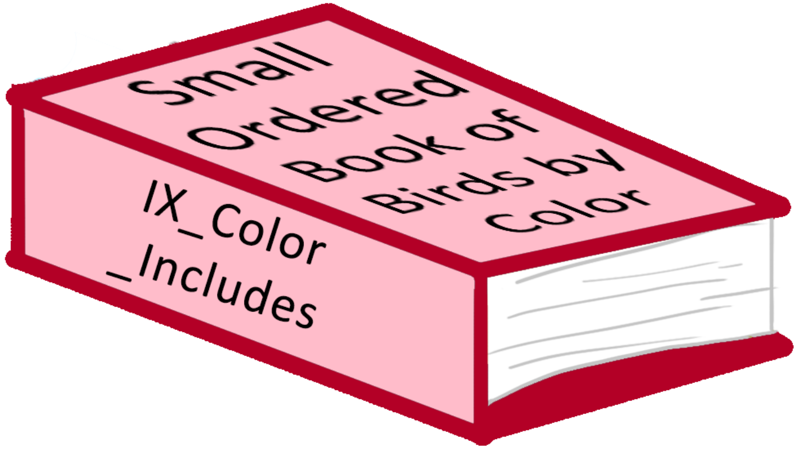
# Nonclustered Index

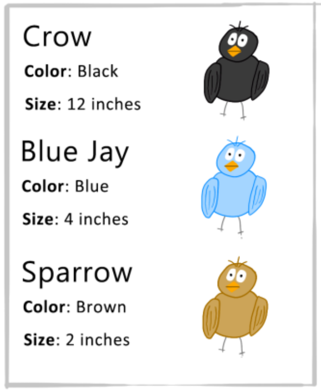
```
CREATE NONCLUSTERED INDEX IX_BirdName_Includes
    ON dbo.FieldGuide (BirdName)
    INCLUDE (Color, Size);
```



Our “included” columns – these are the columns of data that get copied into our index.

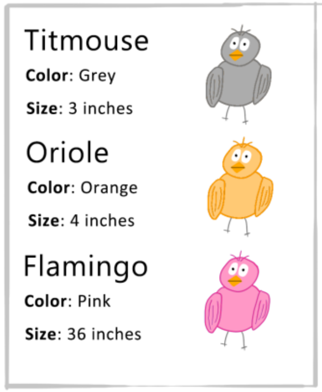
# Nonclustered Index Part 2




Crow
**Color**: Black
**Size**: 12 inches

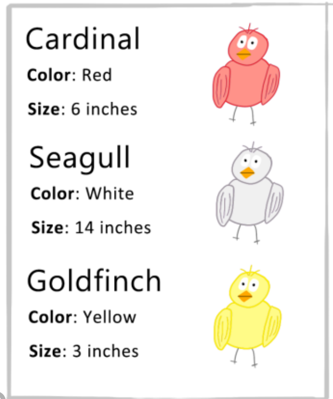
Blue Jay
**Color**: Blue
**Size**: 4 inches

Sparrow
**Color**: Brown
**Size**: 2 inches


Titmouse
**Color**: Grey
**Size**: 3 inches

Oriole
**Color**: Orange
**Size**: 4 inches

Flamingo
**Color**: Pink
**Size**: 36 inches

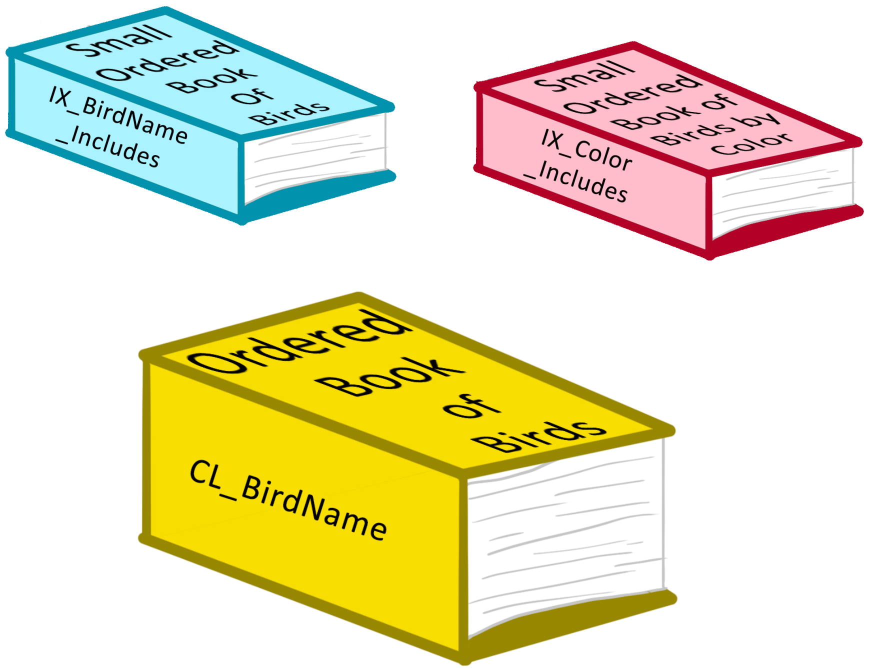

Cardinal
**Color**: Red
**Size**: 6 inches

Seagull
**Color**: White
**Size**: 14 inches

Goldfinch
**Color**: Yellow
**Size**: 3 inches

- Same subset of columns as previous nonclustered index
- Sorted in different order, so better for certain queries (and worse for others)

# Nonclusered Index Part 2



```sql
CREATE NONCLUSTERED INDEX IX_Color_Includes
    ON dbo.FieldGuide (Color)
    INCLUDE (BirdName, Size);
```

# Cardinality

| | Name | Color | Size |
|---|---|---|---|
| 1. | Blue Jay | Blue | Medium |
| 2. | Cardinal | Red | Medium |
| 3. | Crow | Black | Large |
| 4. | Flamingo | Pink | Extra Large |
| 5. | Oriole | Orange | Medium |
| 6. | Sparrow | Brown | Small |

```
SELECT
    Name, Color, Size
FROM
    dbo.Birds
WHERE
    Color = 'Red'
    AND Size = 'Medium'
```

```
CREATE INDEX IX_Size
ON dbo.Birds (Size,Color)
INCLUDE (Name)
```

| | |
|---|---|
| Rows 1-20 | Extra Small |
| Rows 21-40 | Small |
| Rows 41-60 | Medium |
| Rows 61-80 | Large |
| Rows 81-100 | Extra Large |

```
CREATE INDEX IX_Color
ON dbo.Birds (Color,Size)
INCLUDE (Name)
```

| | |
|---|---|
| Rows 1-5 | Amber |
| Rows 6-10 | Black |
| Rows 11-15 | Blue |
| Rows 16-20 | Brown |
| Rows 21-25 | Fuscia |
| Rows 26-30 | Gold |
| Rows 31-35 | Green |
| Rows 36-40 | Grey |
| Rows 41-45 | Hazel |
| Rows 46-50 | Light Blue |
| Rows 51-55 | Mauve |
| Rows 56-60 | Orange |
| Rows 61-65 | Pink |
| Rows 66-70 | Purple |
| Rows 71-75 | Red |
| Rows 76-80 | Scarlet |
| Rows 81-85 | Silver |
| Rows 86-90 | Tangerine |
| Rows 91-95 | White |
| Rows 96-100 | Yellow |

# Summary

Indexes solve two big problems:

1. Sort Your Data
    - JOIN, WHERE, GROUP BY, etc… can utilize this pre-sorted data.
2. Allow for Higher Data Density
    - So there are fewer data pages to read

# Any questions so far?

# QUIZ TIME!

- Does key column order matter?
  - Imagine a book of birds sorted on bird name, will it help me find unknown birds?
    - No – it's better to sort on some other attribute first, such as color.
  - Imagine my book has 100 birds – evenly distributed across 20 distinct colors and 5 different sizes. If I want to find brown birds that are 3" tall, do I first sort on color or on size?
    - Color – it's more selective. Filtering on a single color will leave me with 5 possible bird choices. Filtering on size first would leave me with 20 possible bird choices.
  - Does include column order matter?
    - Include columns are primarily there to prevent your queries from having to go to a different index to get additional columns. Order doesn't matter here.

# Common Indexing Recommendations

- These are suggestions, not absolutes!
- They are good to get you going, but they won’t be the best option in every scenario (especially complex situations).
- But they are good starting points.

# Heap

- Benefits:
    - If no clustered index, it means data can be written quickly wherever there is free space. *Possibly* useful in ETL staging.
- Disadvantages:
    - A well-defined clustered index will almost always beat out a heap.
    - Every operation that is not an insert will be as fast or significantly faster on a clustered index than on a heap.
- Recommendation:
    - Don’t use them.

# Heap

```sql
CREATE TABLE dbo.BirdSpecies
(
    Id bigint IDENTITY,
    CommonName nvarchar(100),
    PrimaryColor nvarchar(30),
    SecondaryColor nvarchar(30),
    HeightInches tinyint
);
```

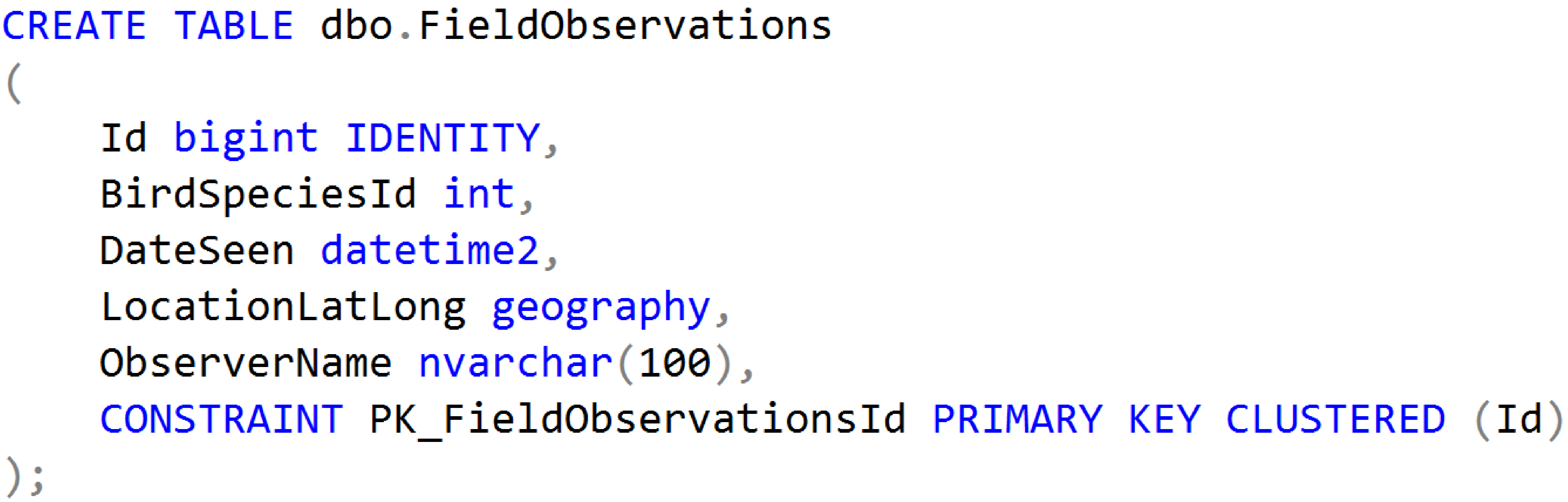
# Clustered Index

- Benefits:
  - Better than a heap 99% of the time.
  - Can make lookups quicker.
- Disadvantages:
  - A poorly defined clustered index could take up extra space and be inefficient.
- Other Notes:
  - You get one per table – choose wisely!
  - The clustering key gets copied to every nonclustered index page…don’t make it wide (either # of columns or byte size of columns)

# Clustered Index

- Recommended keys for a clustered index:
  - **Int/bigint IDENTITY** – will guarantee uniqueness, ever-increasing.
    - If I don’t know what to do, this is my default.
  - **Datetime2** – ever increasing, may not be unique so might want to add a second IDENTITY column as part of the key.
    - This is particularly useful if you are always going to be querying on that datetime2 field.
  - **Sequential GUIDs** – ever increasing, unique across systems
    - Only would use this option **if** need a unique value across tables, servers.

# Clustered Index - Identity

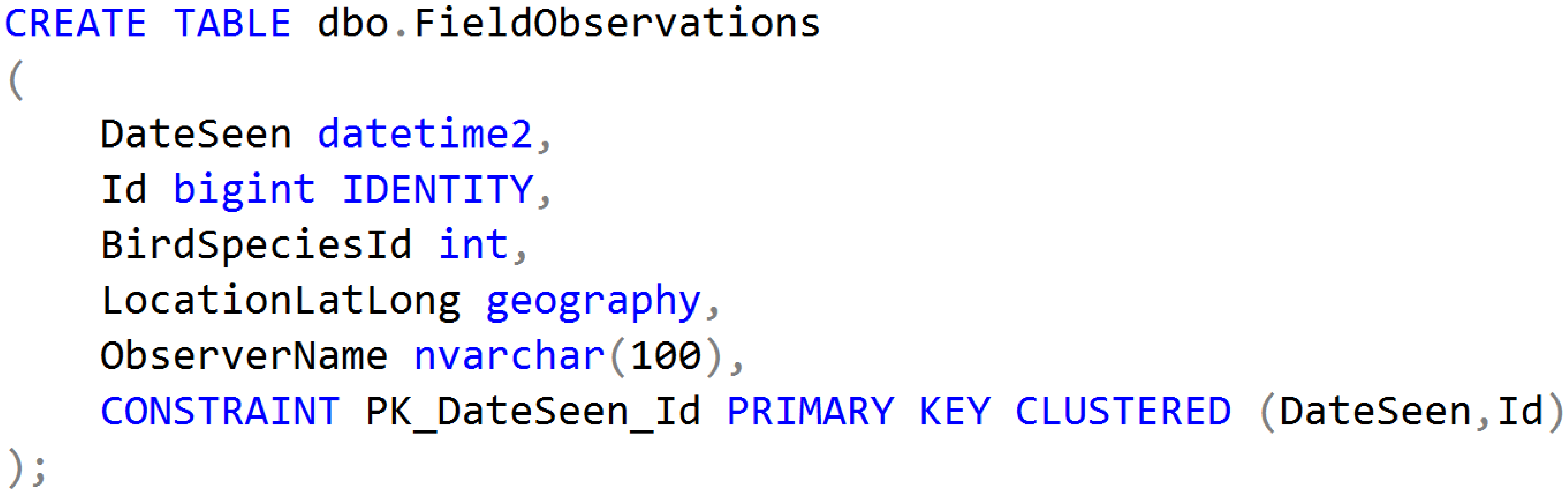


```
CREATE TABLE dbo.FieldObservations
(
    Id bigint IDENTITY,
    BirdSpeciesId int,
    DateSeen datetime2,
    LocationLatLong geography,
    ObserverName nvarchar(100),
    CONSTRAINT PK_FieldObservationsId PRIMARY KEY CLUSTERED (Id)
);
```

# A Quick Note about Primary Keys

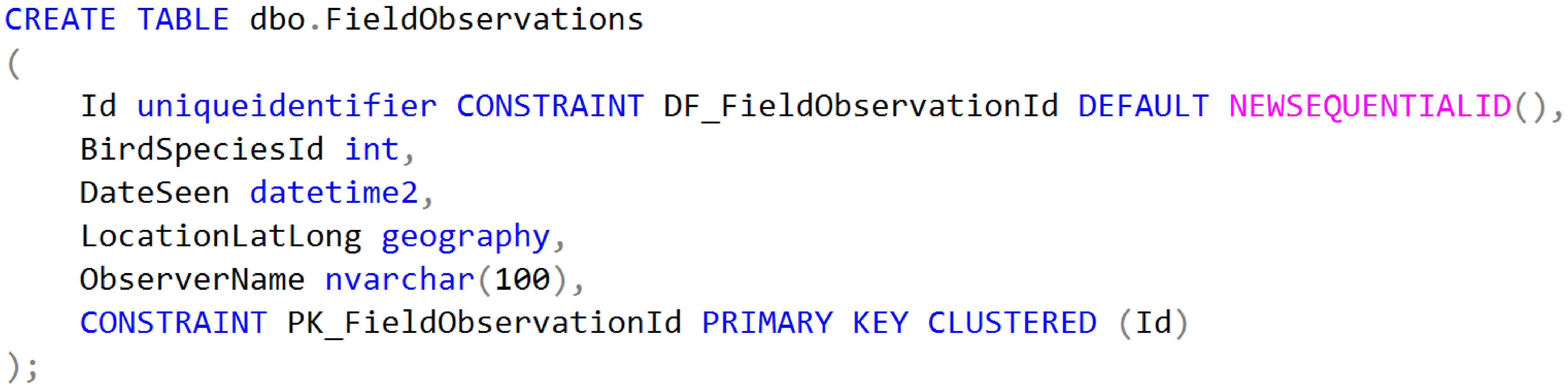
- PKs != Clustered Indexes
- PKs are a constraint that indicate column combination makes a row unique
- You can have PK as a nonclustered index.  Or a clustered index without a PK.  They're independent of each other.

# Clustered Index – Datetime2 + identity



```
CREATE TABLE dbo.FieldObservations
(
    DateSeen datetime2,
    Id bigint IDENTITY,
    BirdSpeciesId int,
    LocationLatLong geography,
    ObserverName nvarchar(100),
    CONSTRAINT PK_DateSeen_Id PRIMARY KEY CLUSTERED (DateSeen,Id)
);
```

# Clustered Index – Sequential GUID



```
CREATE TABLE dbo.FieldObservations
(
    Id uniqueidentifier CONSTRAINT DF_FieldObservationId DEFAULT NEWSEQUENTIALID(),
    BirdSpeciesId int,
    DateSeen datetime2,
    LocationLatLong geography,
    ObserverName nvarchar(100),
    CONSTRAINT PK_FieldObservationId PRIMARY KEY CLUSTERED (Id)
);
```

# Nonclustered Index

- Benefits:
  - Can store data sorted in a different order.
  - Can store data with fewer columns (greater density)
- Disadvantages:
  - Use extra space.
  - Need to be modified on every insert/update/delete.
  - Need to be maintained for fragmentation.

# Nonclustered Index

- Recommended keys for a nonclustered index:
    - **Foreign keys** – fields used to join on (regardless if they have a FK constraint or not)
    - **WHERE predicates** – if a query is filtering on a subset of rows
    - **GROUP BYs** – if using GROUP BYs or window functions, use your grouping/partitioning columns as keys

# Nonclustered Index

```
-- Who observed birds on March 1st?
SELECT
    fo.ObserverName, fo.DateSeen
FROM
    dbo.FieldObservations fo
WHERE
    fo.DateSeen >= '2018-03-01'
    AND fo.DateSeen < '2018-03-02'
```

```
CREATE TABLE dbo.FieldObservations
(
    Id bigint IDENTITY,
    BirdSpeciesId int,
    DateSeen datetime2,
    LocationLatLong geography,
    ObserverName nvarchar(100),
    CONSTRAINT PK_FieldObservationsId PRIMARY KEY CLUSTERED (Id)
);
```

```
CREATE NONCLUSTERED INDEX IX_DateSeen_Includes
ON dbo.FieldObservations (DateSeen) INCLUDE (ObserverName)
```

# Nonclustered Index

```sql
-- Who observed birds on March 1st?
SELECT
    fo.ObserverName, fo.DateSeen
FROM
    dbo.FieldObservations fo
WHERE
    fo.DateSeen >= '2018-03-01'
    AND fo.DateSeen < '2018-03-02'
    AND fo.ObserverName = 'Bert'
```

```sql
CREATE TABLE dbo.FieldObservations
(
    Id bigint IDENTITY,
    BirdSpeciesId int,
    DateSeen datetime2,
    LocationLatLong geography,
    ObserverName nvarchar(100),
    CONSTRAINT PK_FieldObservationsId PRIMARY KEY CLUSTERED (Id)
);
```

```sql
CREATE NONCLUSTERED INDEX IX_DateSeen_ObserverName
ON dbo.FieldObservations (DateSeen,ObserverName)
```

This assumes DateSeen is more selective (has more unique values) than ObserverName

# Nonclustered Index

```
-- How many sightings per observer?
SELECT bs.ObserverName, COUNT(bs.Id) AS ObserverCount
FROM
    dbo.FieldObservations bs
GROUP BY
    bs.ObserverName
```

```
CREATE NONCLUSTERED INDEX IX_ObserverName
ON dbo.FieldObservations (ObserverName)
```

(A one column index is probably not best practice.  Perhaps we can modify this index to be used by other queries as well…)

# Nonclustered Index

```sql
-- How many times have I seen each bird?
SELECT  fo.ObserverName,
        COUNT(DISTINCT fo.BirdSpeciesId) AS BirdCount,
        bs.CommonName
FROM
    dbo.FieldObservations fo
    INNER JOIN dbo.BirdSpecies bs
        ON fo.BirdSpeciesId = bs.Id
WHERE
    fo.ObserverName = 'Bert'
GROUP BY
    fo.ObserverName,
    bs.CommonName
```

```sql
CREATE TABLE dbo.BirdSpecies
(
    Id bigint IDENTITY,
    CommonName nvarchar(200),
    ScientificName nvarchar(200),
    TaxonRank nvarchar(20),
    Kingdom nvarchar(20),
    Phylum nvarchar(20),
    Class nvarchar(20),
    [Order] nvarchar(20),
    Family nvarchar(50),
    Genus nvarchar(50),
    CONSTRAINT PK_BirdSpeciesId PRIMARY KEY CLUSTERED (Id)
);
GO
```

Definitely:

```sql
CREATE NONCLUSTERED INDEX IX_ObserverName_BirdSpeciesId
ON dbo.FieldObservations (ObserverName, BirdSpeciesId)
```

Maybe:

```sql
CREATE NONCLUSTERED INDEX IX_BirdSpeciesId_Includes
ON dbo.BirdSpecies (Id) INCLUDE (CommonName)
```

# Nonclustered Index

```
-- We want a query to search all of our columns
DECLARE @SearchValue nvarchar(20);

SELECT
    bs.Id, bs.CommonName
FROM
    dbo.BirdSpecies bs
WHERE
    bs.ScientificName = @SearchValue
    OR bs.Kingdom = @SearchValue
    OR bs.Phylum = @SearchValue
    OR bs.Class = @SearchValue
    OR bs.[Order] = @SearchValue
    OR bs.Family = @SearchValue
    OR bs.[Genus] = @SearchValue
```

```
CREATE TABLE dbo.BirdSpecies
(
    Id bigint IDENTITY,
    CommonName nvarchar(200),
    ScientificName nvarchar(200),
    TaxonRank nvarchar(20),
    Kingdom nvarchar(20),
    Phylum nvarchar(20),
    Class nvarchar(20),
    [Order] nvarchar(20),
    Family nvarchar(50),
    Genus nvarchar(50),
    CONSTRAINT PK_BirdSpeciesId PRIMARY KEY CLUSTERED (Id)
);
GO
```

One query per column?  Maybe.  But probably too painful.

```
CREATE NONCLUSTERED INDEX IX_ScientificName_Includes
    ON dbo.BirdSpecies (ScientificName) INCLUDE (CommonName);
CREATE NONCLUSTERED INDEX IX_Kingdom_Includes
    ON dbo.BirdSpecies (Kingdom) INCLUDE (CommonName);
CREATE NONCLUSTERED INDEX IX_Phylum_Includes
    ON dbo.BirdSpecies (Phylum) INCLUDE (CommonName);
```

# Nonclustered Index

```
-- We want a query to search all of our columns
-- Might have to unpivot our data first
SELECT
    bs.Id, bs.CommonName
FROM
    dbo.BirdSpecies bs
WHERE
    bs.CategoryValue = @SearchValue
```

```
CREATE TABLE dbo.BirdSpecies
(
    Id bigint IDENTITY,
    CommonName nvarchar(200),
    CategoryName nvarchar(30),
    CategoryValue nvarchar(200)
    CONSTRAINT PK_BirdSpeciesId PRIMARY KEY CLUSTERED (Id)
);
```

```
CREATE NONCLUSTERED INDEX IX_CategoryValue_Include
    ON dbo.BirdSpecies (CategoryValue) INCLUDE (CommonName)
```

Not every problem is an index problem.

Might be better to restructure your table instead.

# Thank you!

SQL WITH BERT

New Episodes Every Tuesday

☺ @bertwagner

☺ bertwagner.com

☺ youtube.com/bertwagner

☺ bert@bertwagner.com

New posts and videos every Tuesday!

# Appendix - When do indexes hurt

- Hotspots - last page is constantly being locked by inserts - this can slow things down.  This may be a time where a clustered index may not want to be on something incremental like identity or date.  GUIDs actually solve some performance issues here *shudder*
- If you have a query that is slow, but runs once per month for reporting, don’t index it.  Either deal with the slow tradeoff, or do something like run a job to create that index then remove it after the job runs.
- Insert/Update/Delete - every time this happens, every index must be modified
- page splits
- fill factor
- fragmentation (even ssds, fragmentation stinks bc still reading more pages into buffer pool
- maintenance - operational burden.  someone needs to run maintenance scripts.  index maintenance scripts can have lots of problems with blocking.  the more indexes you have to manage, the more possibilyt you have with trouble arising.

# Appendix - Other types of indexes

- Filtered, computed column index (storing preparsed data or fixing conversion issues without changing queries), Columnstore (analytical queries), spatial, hierarchical, xml, etc..
- Link to resources