

# **Relational Databases and Datawarehousing Indexes and performance**

# Is performance still relevant?

- On one side: Moore's law
  - Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years. Moore's law is an observation and projection of a historical trend. Rather than a law of physics, it is an empirical relationship linked to gains from experience in production.
  - No longer valid since 2016?
- On the other side: Wirth's law
  - Software is getting slower more rapidly than hardware is becoming faster.

- Anyway ...



- Often indexes offer the solution

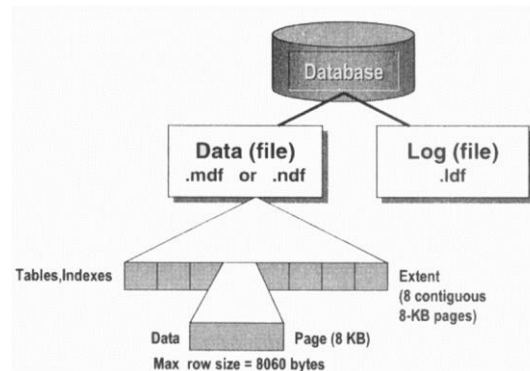


# Space allocation by SQL Server

- SQL Server uses random access files
- Space allocation in extents and pages
- Page = 8 kB block of contiguous space
- Extent = 8 logical consecutive pages.
  - uniform extents: for one db object
  - mixed extents: can be shared by 8 db objects (=tables, indexes)
- New table or index: allocation in mixed extent
- Extension > 8 pages: in uniform extent
- <https://techyaz.com/sql-server/understanding-sql-server-data-files-pages-extents/>

Database name: xtreme  
Owner:   
☒ Use full-text indexing

Logical Name	File Type	Filegroup	Initial Size (MB)
Xtreme_Data	ROWS Data	PRIMARY	14
Xtreme_Log	LOG	Not Applicable	32

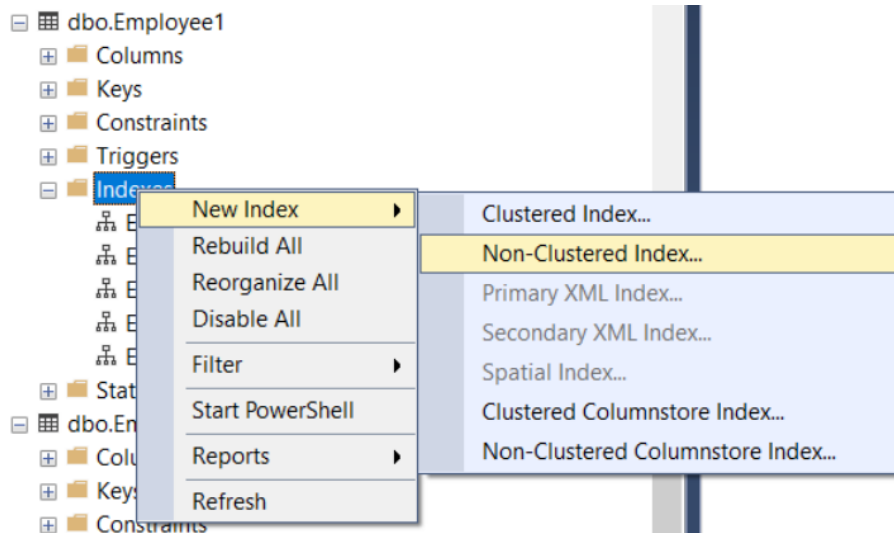


# Creation of indexes

```
CREATE [UNIQUE] [ | NONCLUSTERED]  
INDEX index_name ON table (kolom [,...n])
```

*create index*

- unique index: all values in the indexed column should be unique
- remark:
  - when defining an index the table can be empty or filled.
  - columns in a unique index should have the not null constraint



# Removing indexes

```
DROP INDEX table_name.index [,...n]
```

*deleting index*

- unique all values in the indexed column should be unique
- remark:
  - when defining an index the table can be empty or filled.
  - columns in a unique index should have the not null constraint

# Clustered vs. Non-clustered indexes

- See the short version (2min)  
[https://www.youtube.com/watch?v=AINh6\\_LqnDM](https://www.youtube.com/watch?v=AINh6_LqnDM)
- See the long version (6min)  
<https://www.youtube.com/watch?v=ITcOiLSfVJQ>



# Table scan

- Heap: unordered collection of data-pages without clustered index= default storage of a table
- Access via Index Allocation Map (IAM)
- Table scan: if a query fetches all pages of the table often best to avoid!
- Other performance issues with heap:
  - fragmentation: table is scattered over several, non-consecutive pages
  - forward pointers: if a variable length row (e.g. varchar fields) becomes longer upon update, a forward pointer to another page is added. → table scan even slower

# Table scan

- 1<sup>st</sup> step: put unordered data in tables
- Very inefficient: if we are looking for a phonenummer, we would have to go through all the phonenumbers, even when we found a match, because there are perhaps more matches later on
- Table scans aren't always a bad thing, e.g. when you have to retrieve a lot of data out of the table



The diagram illustrates a table scan process using four document icons, each containing a list of names and phone numbers. The icons are arranged horizontally, with an ellipsis between the third and fourth, indicating a continuous sequence of data blocks. Each icon has a folded top-right corner, suggesting a document or page.

Alexander, Mary 344-555-0133	Martinez, Frank 171-555-0147	Kitt, Sandra 303-555-0117	Clayton, Jane 206-555-0195
Kurtz, Jeffrey 452-555-0179	Haines, Betty 867-555-0114	Brewer, Alan 494-555-0134	Johnson, Brian 320-555-0134
Vessa, Robert 560-555-0171	Burnett, Linda 121-555-0121	Campbell, Frank 491-555-0132	Liu, David 440-555-0132
Thames, Judy 799-555-0118	Harris, Keith 170-555-0127	Logan, Todd 783-555-0110	Diaz, Brenda 147-555-0192

# Table scan

- In a phone book: data is ordered by last name and if there are duplicates, data is also ordered by first name
- => 2<sup>nd</sup> step: You can try to put the data in physical order
- Doesn't solve the performance problem. It would do a similar type of table scan, but this time it would know where to stop because the table data is ordered.

Adams, Jay 158-555-0142	Burnett, Linda 121-555-0121	Diaz, Brenda 147-555-0192	Taylor, Mike 204-555-0189
Alexander, Mary 344-555-0133	Campbell, Frank 491-555-0132	Doyle, Patricia 899-555-0134	Thames, Judy 799-555-0118
Benson, Edna 789-555-0189	Clayton, Jane 206-555-0195	Evans, John 581-555-0172	Vargas, Gary 112-555-0176
Brewer, Alan 494-555-0134	Cooper, Scott 733-555-0182	Haines, Betty 867-555-0114	Vessa, Robert 560-555-0171

# Clustered index

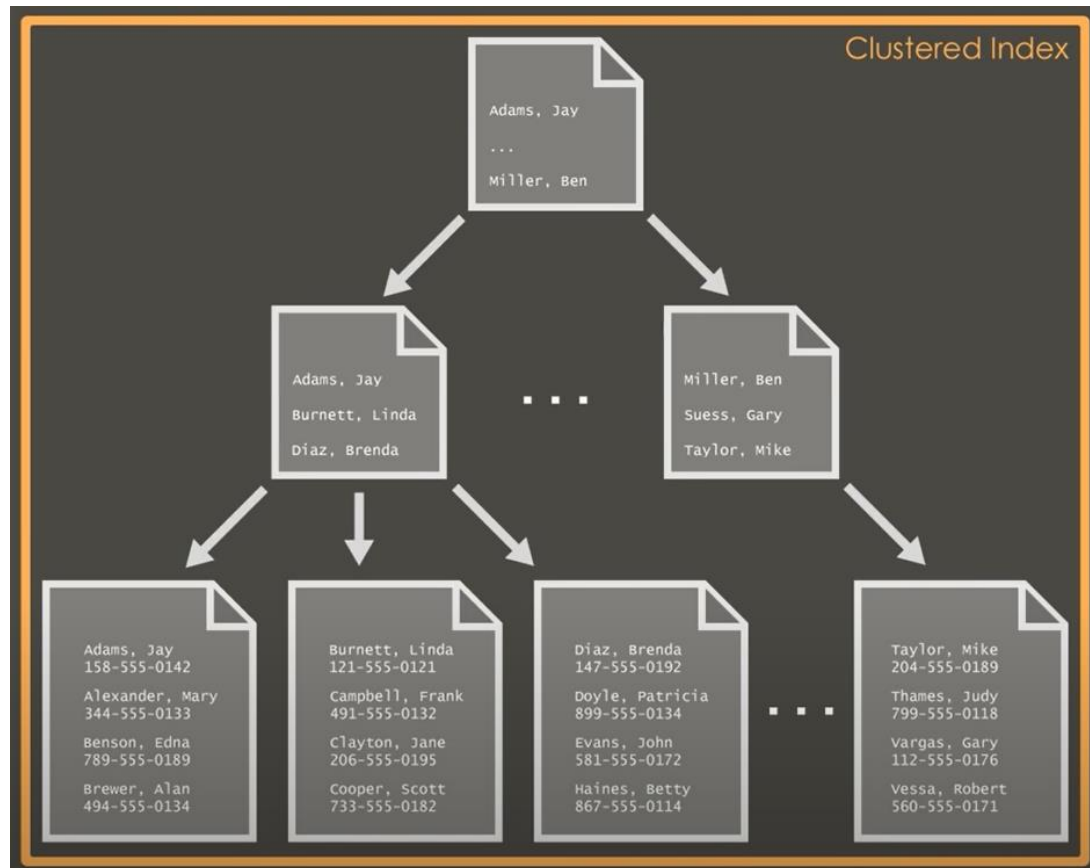
- After the table data is put into physical order, SQL Server builds up a set of index pages that allows queries to navigate directly to the data they're interested in. The entire structure, including the base table data, is called a clustered index. When a query navigates through the clustered index tree to the base table data, this is called a clustered index seek.

# Clustered index

```
CREATE TABLE dbo.PhoneBook
```

```
(  
  LastName varchar(50) NOT NULL,  
  FirstName varchar(50) NOT NULL,  
  PhoneNumber varchar(50) NOT NULL  
);
```

```
CREATE CLUSTERED INDEX IX_PhoneBook_CI  
ON dbo.PhoneBook(LastName, FirstName);
```



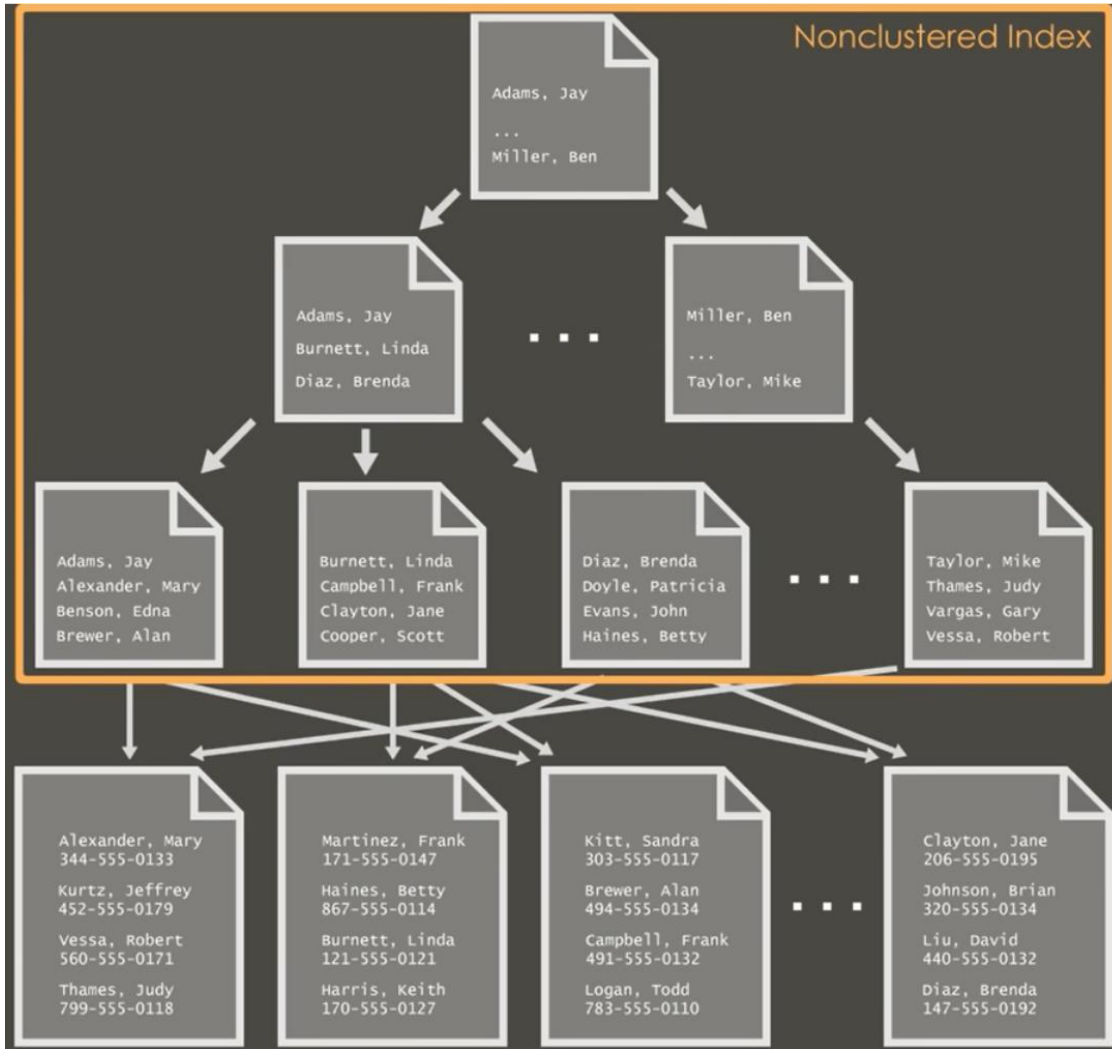
## Non Clustered index

- Since a clustered index contains the base table data itself, you can only create one clustered index.
- You have to create a separate physical structure that has the same index tree like the clustered index. This time, though, instead of having the base data at the bottom (or leaf) level of the tree, there is a set of pointers (or references), back to the base data => we can use any index key order

# Non Clustered index

```
CREATE TABLE dbo.PhoneBook  
(  
    LastName varchar(50) NOT NULL,  
    FirstName varchar(50) NOT NULL,  
    PhoneNumber varchar(50) NOT NULL  
);
```

```
CREATE NONCLUSTERED INDEX IX_PhoneBook_NCI  
ON dbo.PhoneBook(LastName, FirstName);
```



## Non Clustered index

- In this example, the base data is a heap, and the references back to it are RIDs (row identifiers). These are the physical locations of the rows in the table.
- Since a non clustered index is separate from the base data, the base data could exist instead as a clustered index. If so, the references back to it aren't RIDs, but are the clustered index key values



## Non Clustered index

- Like a clustered index scan and seek, non clustered indexes can have the same operations performed on them. But the data directly available may be limited because usually non clustered indexes only include a subset of column from the table. If values from columns not in the index are requested, the query may navigate back to the base data using the references.

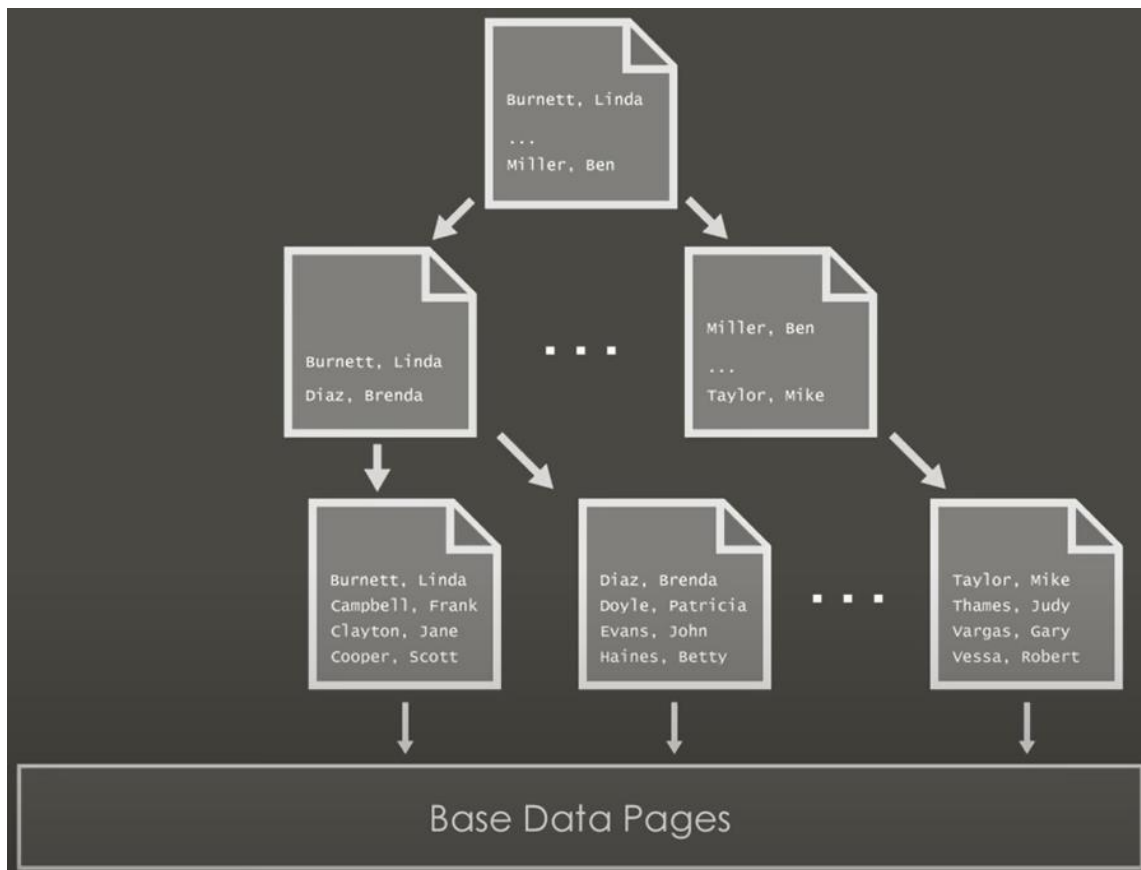
## Filtered index

- Filtered indexes only contain rows that meet a user-defined predicate, and to create these, you have to add a WHERE clause to the index definition.
- A clustered index can't be filtered because it has to contain all the data in the table.

# Filtered index

```
CREATE TABLE dbo.PhoneBook  
(  
    LastName varchar(50) NOT NULL,  
    FirstName varchar(50) NOT NULL,  
    PhoneNumber varchar(50) NOT NULL  
);
```

```
CREATE NONCLUSTERED INDEX IX_PhoneBook_NCI  
ON dbo.PhoneBook(LastName, FirstName)  
WHERE (LastName >= 'Burnett');
```



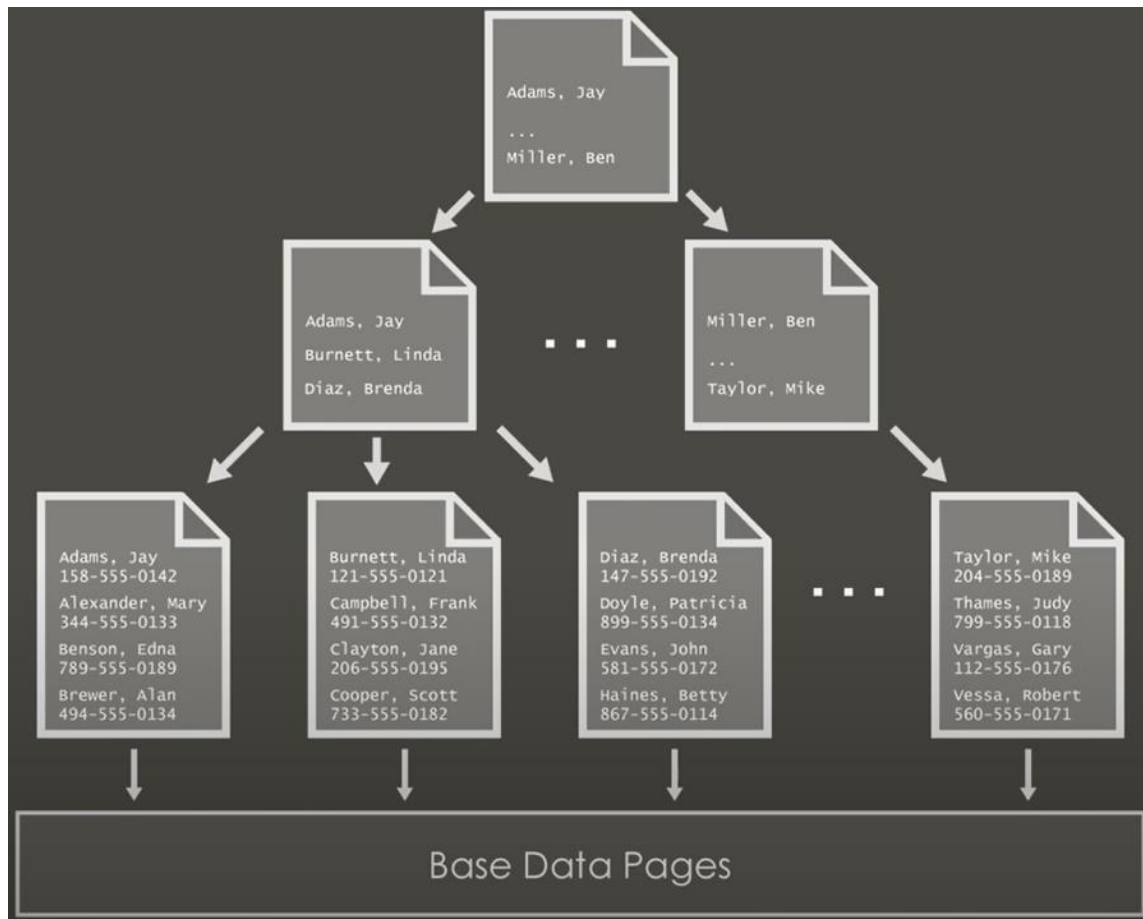
# INCLUDE

- INCLUDE columns add copies of non-key column values to the leaf level of the index tree.
- This means that queries using the nonclustered index won't have to incur the expense of navigating back to the base data to get those non-key column values.
- A clustered index doesn't need INCLUDE columns, because all the columns are available in the leafs.

# INCLUDE

```
CREATE TABLE dbo.PhoneBook  
(  
    LastName varchar(50) NOT NULL,  
    FirstName varchar(50) NOT NULL,  
    PhoneNumber varchar(50) NOT NULL  
);
```


```
CREATE NONCLUSTERED INDEX IX_PhoneBook_NCI  
ON dbo.PhoneBook(LastName, FirstName)  
INCLUDE(PhoneNumber);
```



## Clustered vs Non Clustered index

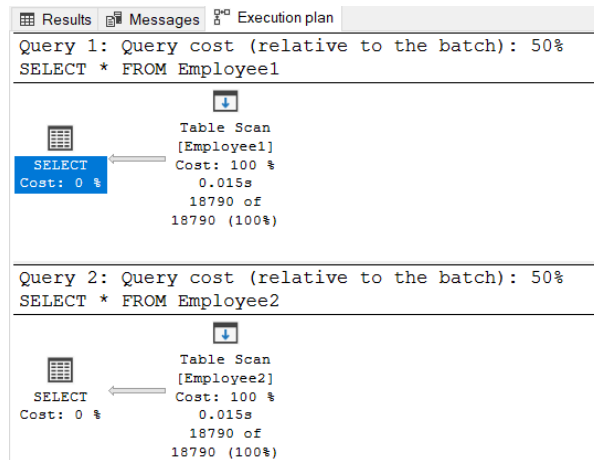
- The clustered index is a way of representing the base data as a whole.
- A non clustered index is a physically separate structure that references the base data and it can have a different sort order.

# Does my query use a table scan?

- Execute the scripts Employee1Idx and Employee2Idx to add the new tables Employee1 and Employee2  
Both tables have about 20 000 records
- Click on the button  before executing the query or choose Query > Include Actual Execution Plan

```
SELECT * FROM Employee1
```

```
SELECT * FROM Employee2
```

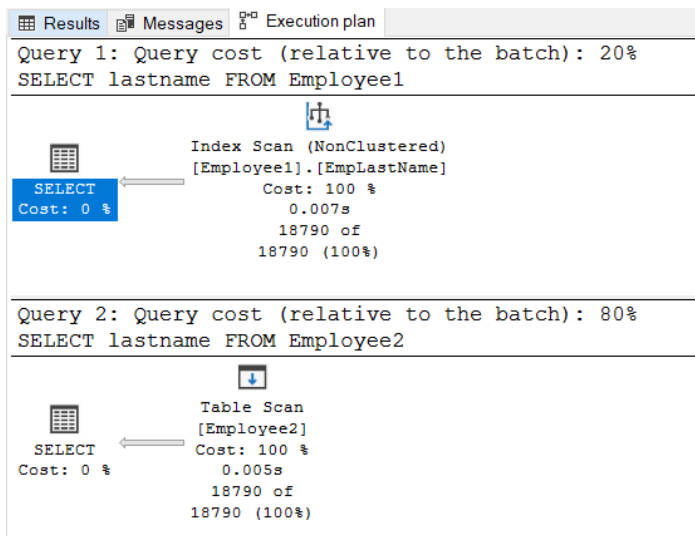


# Does my query uses a table scan?

- Table Employee2 is a copy of Employee1, but without indexes
- Query on Employee2 takes 5x longer!

```
SELECT lastname FROM Employee1
```

```
SELECT lastname FROM Employee2
```





# What is the difference? Indexes!

- **What?**
  - ordered structure imposed on records from a table
  - Fast access through tree structure (B-tree = balanced tree)
- **Why?**
  - can speed up data retrieval
  - can force unicity of rows
- **Why not?**
  - indexes consume storage (overhead)
  - Indexes can slow down updates, deletes and inserts because indexes have to be updated too

# Indexes: library analogy

Consider a card catalog in a library. If you wanted to locate a book named Effective SQL, you would go to the catalog and locate the drawer that contains cards for books starting with the letter E (maybe it will actually be labeled D–G). You would then open the drawer and flip through the index cards until you find the card you are looking for. The card says the book is located at 601.389, so you must then locate the section somewhere within the library that houses the 600 class. Arriving there, you have to find the bookshelves holding 600–610. After you have located the correct bookshelves, you have to scan the sections until you get to 601, and then scan the shelves until you find the 601.3XX books before pinpointing the book with 601.389

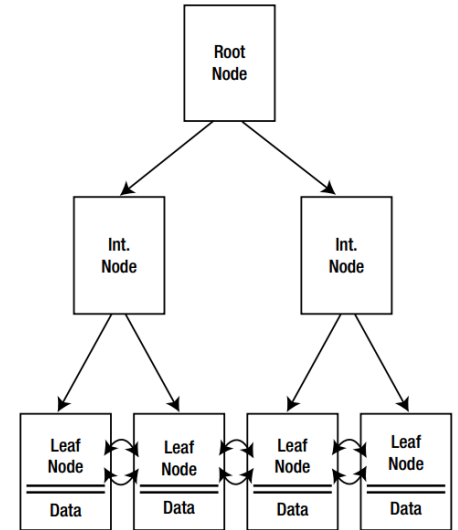
In an electronic database system, it is no different. The database engine needs to first access its index on data, locate the index page(s) that contains the letter E, then look within the page to get the pointer back to the data page that contains the sought data. It will jump to the address of the data page and read the data within that page(s). Ergo, an index in a database is just like the catalog in a library. Data pages are just like bookshelves, and the rows are like the books themselves. The drawers in the catalog and the bookshelves represent the B-tree structure for both index and data pages

# SQL Optimizer

- SQL Optimizer: module in each DBMS
- Analyses and rephrases each SQL command sent to the DB
- Decides optimum strategy for e.g. index use, based on statistics about table size, table use and data distribution
- In SQL searching is used for fields in *where*, *group by*, *having* and *order by* clauses and for fields that are *joined*

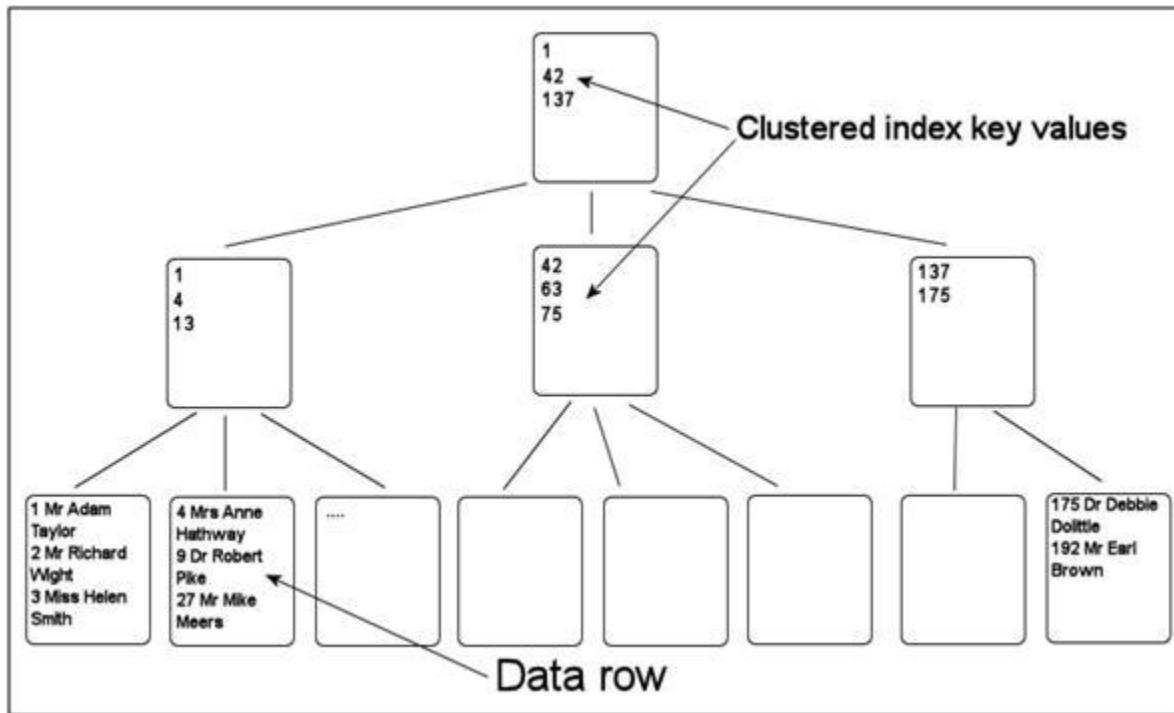
# Clustered index

- The physical order of the rows in a table corresponds to the order in the clustered index.
- As a consequence, each table can have only one clustered index.
- The clustered index imposes unique values and the primary key constraint
- Advantages as opposed to table scan:
  - double linked list ensures order when reading sequential records + no forward pointers necessary



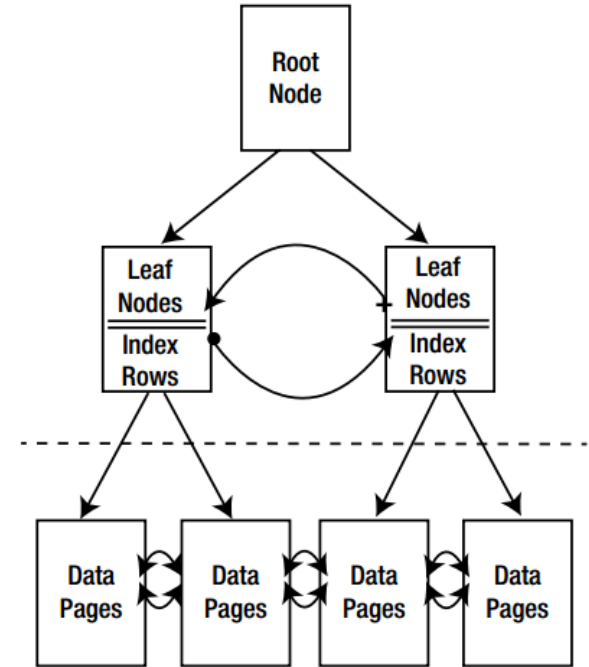
Int. Node = intermediate (tussenliggende) node

# Clustered index



# Non Clustered index

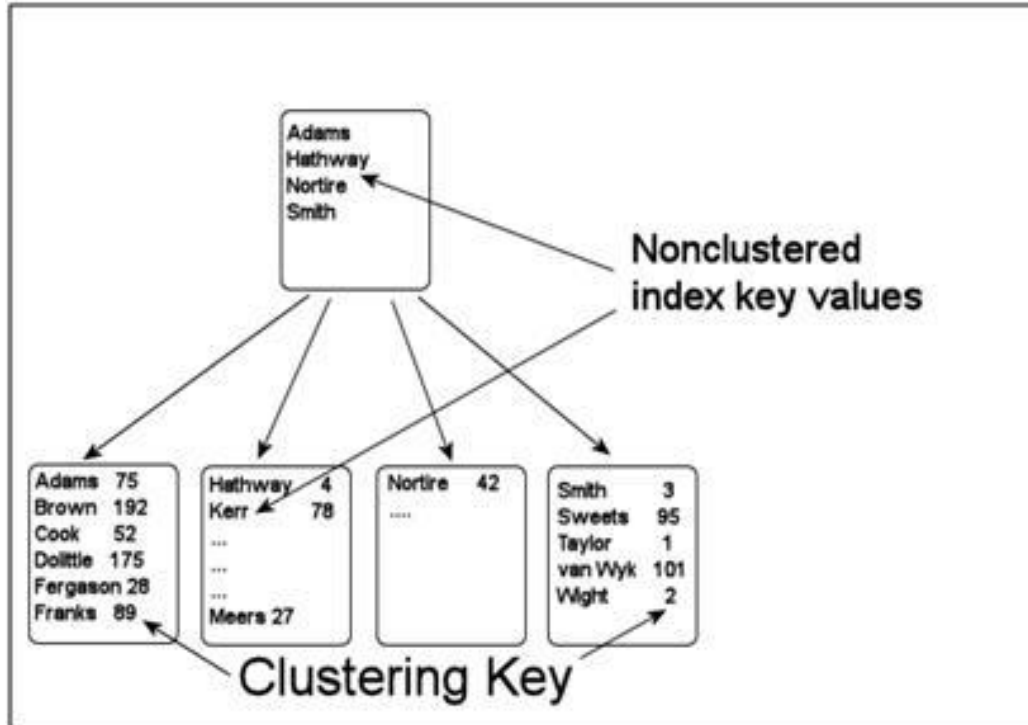
- default index
- slower than clustered index
- > 1 per table allowed
- Forward and backward pointers between leaf nodes
- each leaf contains key value and row locator
  - to position in clustered index if it exists
  - otherwise to heap



## Non Clustered index

- If query needs more fields than present in index, these fields have to be fetched from data pages.
- When reading via non-clustered index:
  - either:  
RID lookup = bookmark lookups to the heap using RID's (= row identifiers)
  - Or:  
key lookup = bookmark lookups to a clustered index, if present

# Non Clustered index



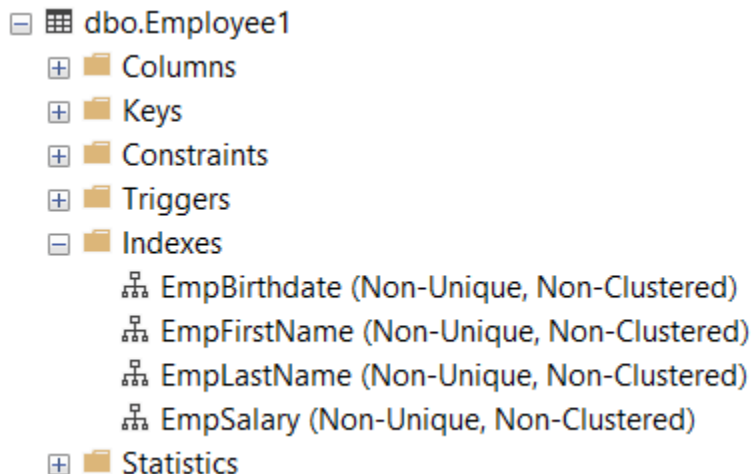


## Covering index

- If a non clustered index not completely covers a query, SQL Server performs a lookup for each row to fetch the data
- Covering index = non-clustered index containing all columns necessary for a certain query
- With SQL Server you can add extra columns to the index (although those columns are not indexed!)

# Covering index

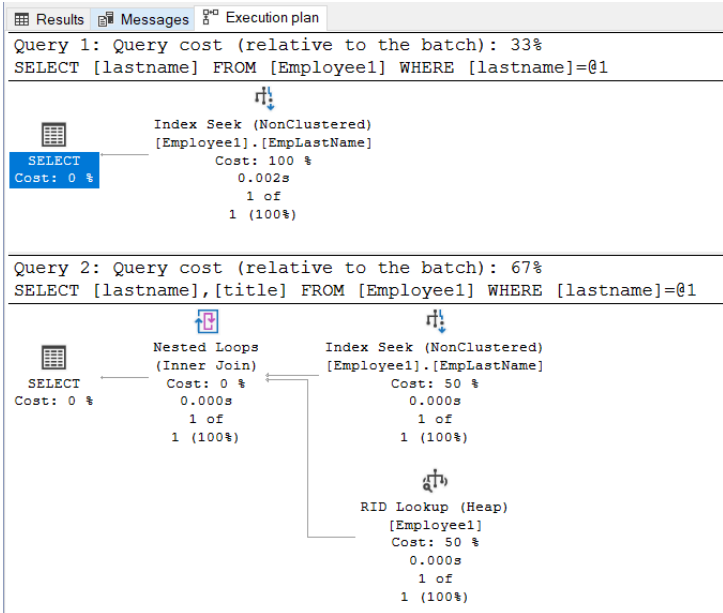
- Current indexes on table Employee1: each index indexes a single field



# Covering index

```
SELECT lastname FROM Employee1 WHERE lastname = 'Duffy'
```

```
SELECT lastname, title FROM Employee1 WHERE lastname = 'Duffy'
```



- Index seek via nonclustered index for seeking lastname = 'Duffy'
- Key loop up in clustered index (= data) for fetching title (not in index)

# Covering index

- Solution: covering index via INCLUDE

```
create nonclustered index EmpLastName_Incl_Title  
ON Employee1(lastname) INCLUDE (title);
```

# Covering index

```
SELECT lastname FROM Employee1 WHERE lastname = 'Duffy'
```


```
SELECT lastname, title FROM Employee1 WHERE lastname = 'Duffy'
```


Results Messages Execution plan

Query 1: Query cost (relative to the batch): 50%

```
SELECT [lastname] FROM [Employee1] WHERE [lastname]=@1
```

---


 Index Seek (NonClustered)  
 [Employee1].[EmpLastName\_Inc...  
 Cost: 100 %  
 0.000s  
 1 of



 SELECT  
 Cost: 0 %


---

Query 2: Query cost (relative to the batch): 50%

```
SELECT [lastname],[title] FROM [Employee1] WHERE [lastname]=@1
```

---


 Index Seek (NonClustered)  
 [Employee1].[EmpLastName\_Inc...  
 Cost: 100 %  
 0.000s  
 1 of  
 1 (100%)


 SELECT  
 Cost: 0 %

# 1 index with several columns vs. several indexes with 1 column

```
CREATE NONCLUSTERED INDEX EmpLastName ON Employee1(lastname);
```

```
+
```

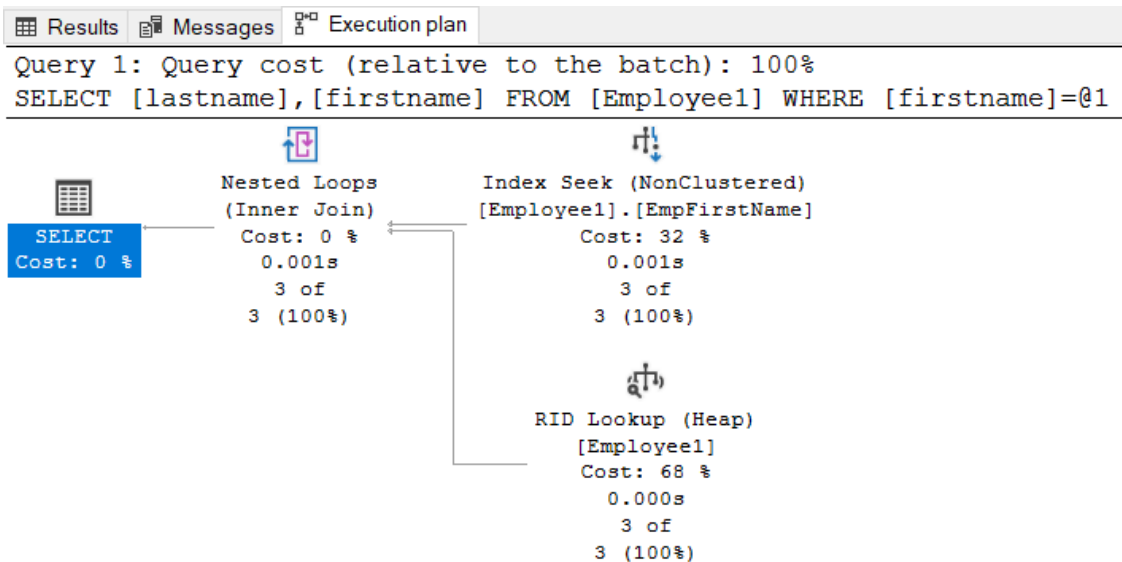
```
CREATE NONCLUSTERED INDEX EmpFirstname ON Employee1(firstname);
```

**OR?**

```
CREATE NONCLUSTERED INDEX EmpLastNameFirstname ON Employee1(lastname, firstname);
```

# 1 index with several columns vs. several indexes with 1 column

```
SELECT lastname, firstname FROM Employee1  
WHERE firstname = 'Chris';
```



## **1 index with several columns vs. several indexes with 1 column**

- When querying (e.g. in WHERE-clause) only 2nd and or 3th, ... field of index, the index is not used. This directly follows from the B-tree table structure of the composed index
- Make your indexes according to the most commonly used queries.



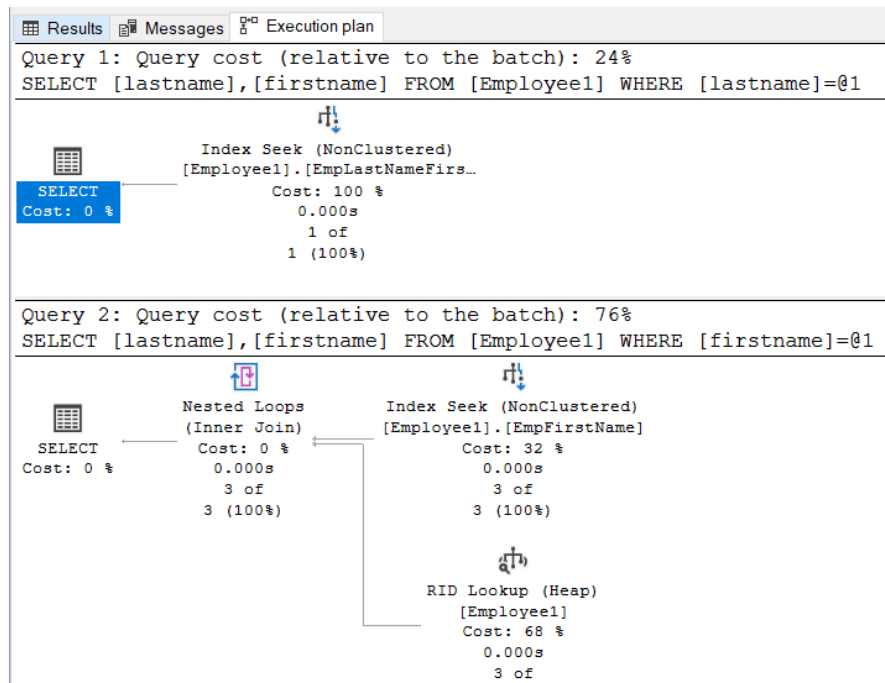
# 1 index with several columns vs. several indexes with 1 column

```
-- Test: Only combined index on lastname and  
firstname
```

```
DROP INDEX EmpLastName ON Employee1;
```

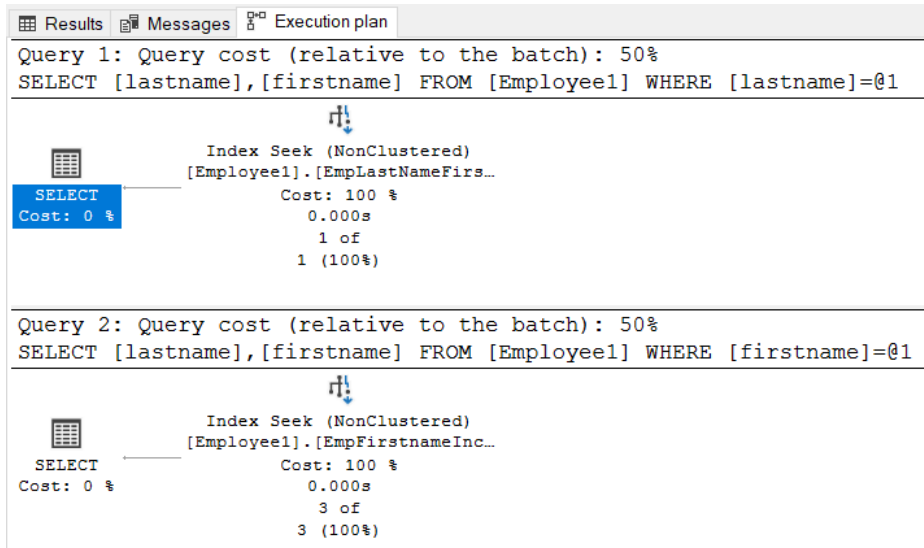
```
SELECT lastname, firstname FROM Employee1  
WHERE lastname = 'Preston'
```

```
SELECT lastname, firstname FROM Employee1  
WHERE firstname = 'Chris';
```



# 1 index with several columns vs. several indexes with 1 column

```
-- With extra index on firstname and covering of  
lastname  
create nonclustered index EmpFirstnameIncLastname  
ON employee1(firstname)  
INCLUDE (lastname);  
  
SELECT lastname, firstname FROM Employee1  
WHERE lastname = 'Preston'  
  
SELECT lastname, firstname FROM Employee1  
WHERE firstname = 'Chris';
```

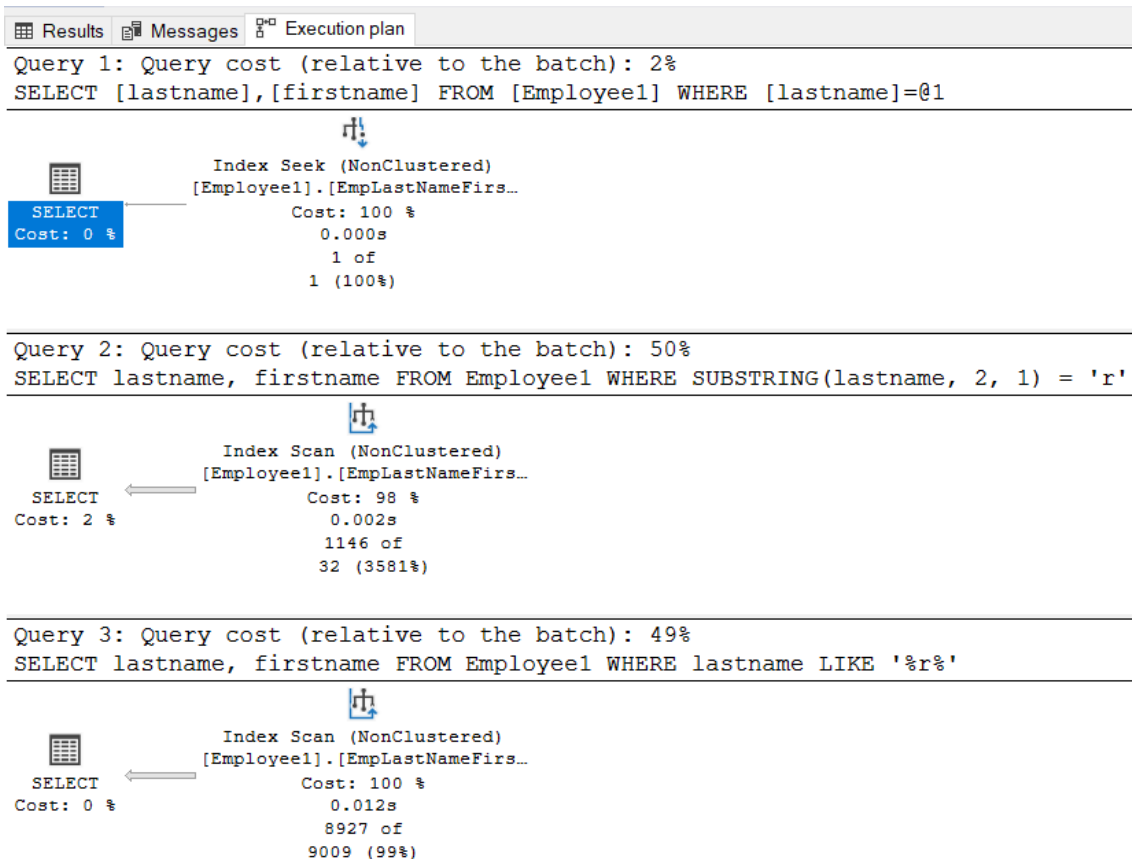


# Use of indexes with functions and wildcards

```
SELECT lastname, firstname  
FROM Employee1  
WHERE lastname = 'Preston'
```

```
SELECT lastname, firstname  
FROM Employee1  
WHERE SUBSTRING(lastname, 2, 1) = 'r'
```

```
SELECT lastname, firstname  
FROM Employee1  
WHERE lastname LIKE '%r%'
```



# Index seek vs Index scan

- Index Seek: tree structure of index is used, resulting in very fast data retrieval.
- Index Scan: index is used but it is scanned from the start till the searched records are found.

# When to use an index?

- Which columns should be indexed?
  - primary and unique columns are indexed automatically
  - foreign keys often used in joins
  - columns often used in search conditions (WHERE, HAVING, GROUP BY ) or in joins
  - columns often used in the ORDER BY clause

# When to use an index?

- Which columns should be not indexed?
  - columns that are rarely used in queries
  - columns with a small number of possible values (e.g. gender)
  - columns in small tables
  - columns of type bit, text or image

# Tips & Tricks: (1) Avoid the use of functions

-- BAD

```
SELECT FirstName, LastName, Birthdate
FROM Employee1
WHERE Year(BirthDate) = 1980;
```

-- GOOD

```
SELECT FirstName, LastName, Birthdate
FROM Employee1
WHERE BirthDate >= '1980-01-01' AND
BirthDate < '1981-01-01';
```

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 95%

```
SELECT lastname, firstname, birthdate FROM Employee1 WHERE Year(BirthDate) = 1980
```

Index Scan (NonClustered)  
[Employee1].[EmpBirthdate]

Cost: 98 %  
0.008s  
513 of  
460 (111%)

SELECT  
Cost: 2 %

Query 2: Query cost (relative to the batch): 5%

```
SELECT [lastname],[firstname],[birthdate] FROM [Employee1] WHERE [BirthDate]>=@1 AND [BirthDate]<@2
```

Index Seek (NonClustered)  
[Employee1].[EmpBirthdate]

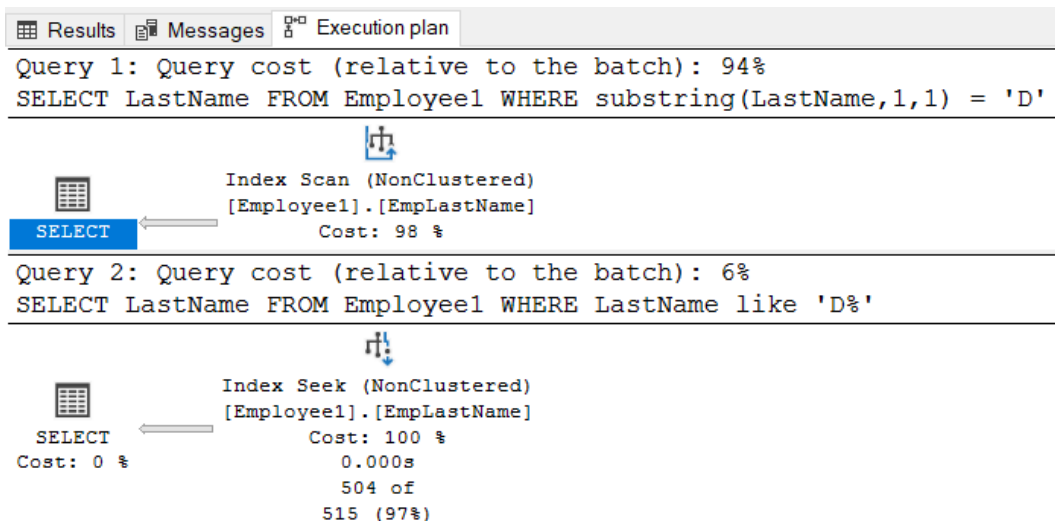
Cost: 100 %  
0.000s  
513 of  
523 (98%)

SELECT  
Cost: 0 %

## Tips & Tricks: (2) Avoid the use of functions

```
-- BAD
SELECT LastName
FROM Employee1
WHERE substring(LastName,1,1) = 'D';
```

```
-- GOOD
SELECT LastName
FROM Employee1
WHERE LastName like 'D%';
```





## Tips & Tricks: (3) avoid calculations, isolate columns

```
-- BAD
SELECT EmployeeID, FirstName, LastName
FROM Employee1
WHERE Salary*1.10 > 100000;

-- GOOD
SELECT EmployeeID, FirstName, LastName
FROM Employee1
WHERE Salary > 100000/1.10;
```

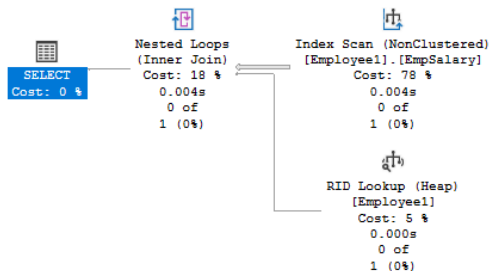
# Tips & Tricks: (3) avoid calculations,

## isolate columns

Results Messages Execution plan

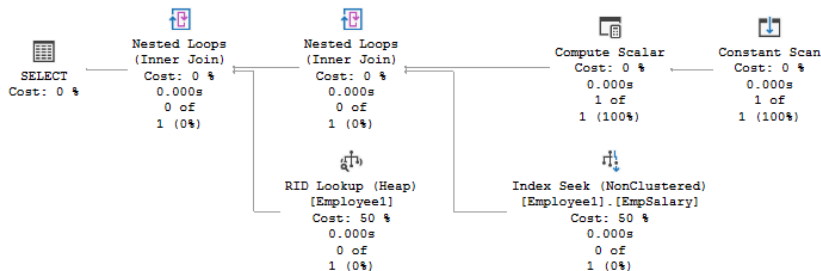
Query 1: Query cost (relative to the batch): 92%

SELECT [EmployeeID],[FirstName],[LastName] FROM [Employee1] WHERE [Salary]\*@1>@2



Query 2: Query cost (relative to the batch): 8%

SELECT [EmployeeID],[FirstName],[LastName] FROM [Employee1] WHERE [Salary]>@1/@2



### Key lookup:

The non-clustered index EmpSalary, holds in each leaf a reference to the location of the total record in the clustered index. Following this reference is called “key lookup”.

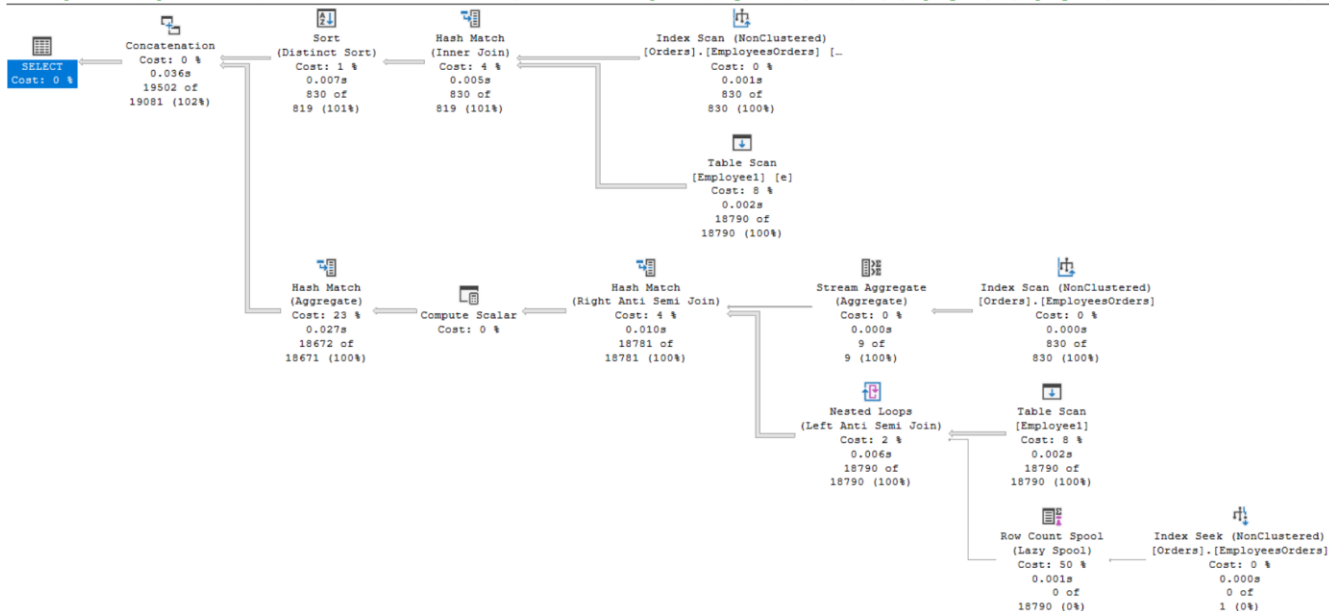
## Tips & Tricks: (4) prefer OUTER JOIN above UNION

```
-- BAD
SELECT lastname, firstname, orderid
from Employee1 e join Orders o on e.EmployeeID = o.employeeid
union
select lastname, firstname, null
from Employee1
where EmployeeID not in (select EmployeeID from Orders)

-- GOOD
SELECT lastname, firstname, orderid
from Employee1 e left join Orders o on e.EmployeeID = o.employeeid;
```

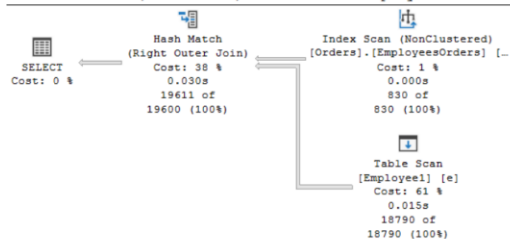
Query 1: Query cost (relative to the batch): 89%

SELECT lastname, firstname,orderid from Employee1 e join Orders o on e.EmployeeID = o.employeeid union select lastname, firstname, null from Employee1 where  
Missing Index (Impact 11.6844): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Employee1] ([EmployeeID]) INCLUDE ([LastName],[Fir



Query 2: Query cost (relative to the batch): 11%

SELECT lastname, firstname,orderid from Employee1 e left join Orders o on e.EmployeeID = o.employeeid



## Tips & Tricks: (5) avoid ANY and ALL

```
-- BAD
SELECT lastname, firstname, birthdate
from Employee1
where BirthDate >= all(select BirthDate from Employee1)

-- GOOD
SELECT lastname, firstname, birthdate
from Employee1
where BirthDate = (select max(BirthDate) from Employee1)
```

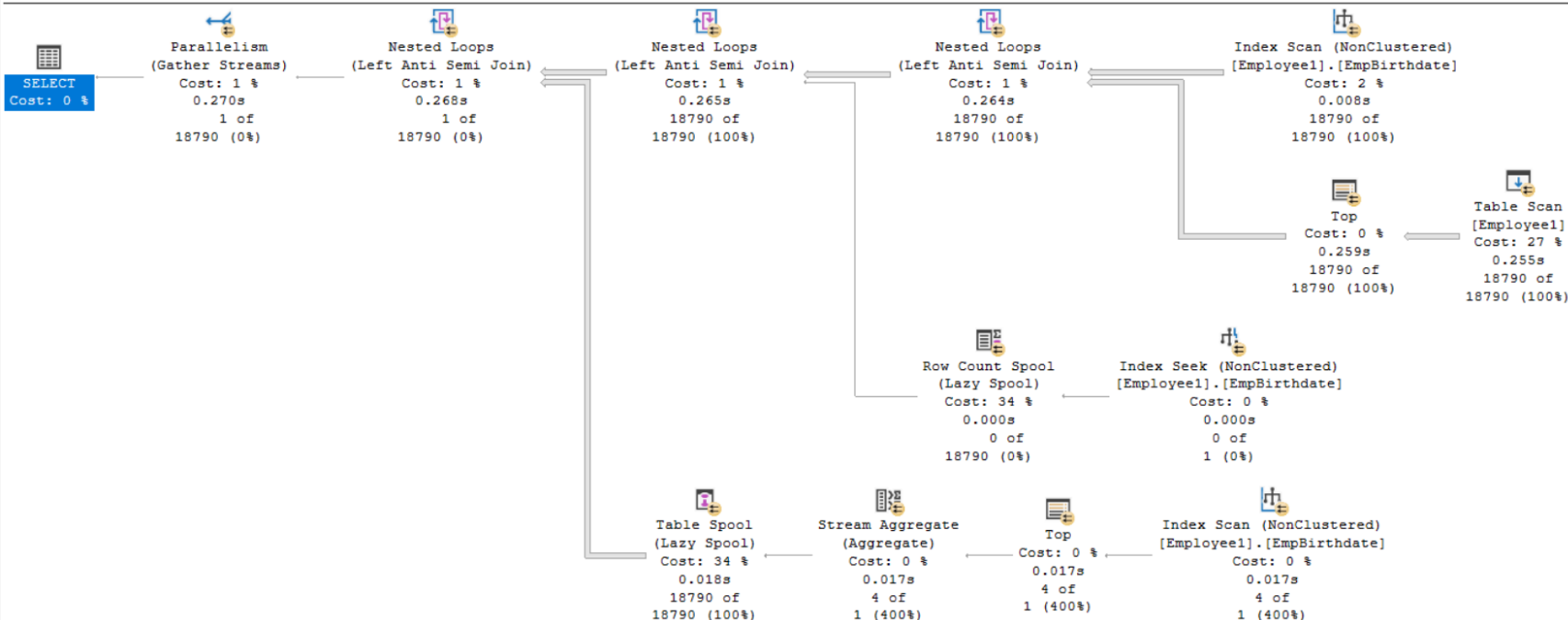
## Tips & Tricks: (5) avoid ANY and ALL

```
-- BAD
SELECT lastname, firstname, birthdate
from Employee1
where BirthDate >= all(select BirthDate from Employee1)

-- GOOD
SELECT lastname, firstname, birthdate
from Employee1
where BirthDate = (select max(BirthDate) from Employee1)
```

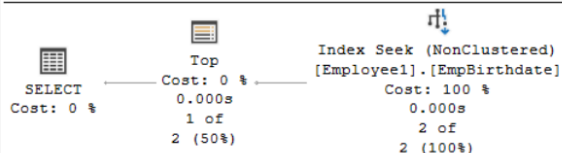
Query 1: Query cost (relative to the batch): 100%

SELECT lastname, firstname, birthdate from Employee1 where BirthDate >= all(select BirthDate from Employee1)



Query 2: Query cost (relative to the batch): 0%

SELECT lastname, firstname, birthdate from Employee1 where BirthDate = (select max(BirthDate) from Employee1)



## Quiz 1 / 5

- Is the following index a good fit for the query?
  - A: Good fit: No need to change anything
  - B: Bad fit: Changing the index or query could improve performance

```
CREATE INDEX tbl_idx ON tbl (date_column);
```

```
SELECT * FROM tbl
```

```
WHERE YEAR(date_column) = 2017;
```



## Quiz 2 / 5

- Is the following index a good fit for the query?
  - A: Good fit: No need to change anything
  - B: Bad fit: Changing the index or query could improve performance

```
CREATE INDEX tbl_idx ON tbl (a, date_column);
```

```
SELECT TOP 1 * FROM tbl  
WHERE a = 12  
ORDER BY date_column DESC;
```

## Quiz 3 / 5

- Is the following index a good fit for the query?
  - A: Good fit: No need to change anything
  - B: Bad fit: Changing the index or query could improve performance

```
CREATE INDEX tbl_idx ON tbl (a, b);
```

```
SELECT * FROM tbl  
WHERE a = 123 AND b = 1;
```

```
SELECT * FROM tbl WHERE b = 123;
```

## Quiz 4 / 5

- Is the following index a good fit for the query?
  - A: Good fit: No need to change anything
  - B: Bad fit: Changing the index or query could improve performance

```
CREATE INDEX tbl_idx ON tbl (text);
```

```
SELECT * FROM tbl  
WHERE text LIKE 'TJ%';
```

## Quiz 5 / 5

- First consider the following index and query:
- Let's say this query returns at least a few rows.

To implement a new functional requirement, another condition ( $b = 1$ ) is added to the WHERE clause. How will the change affect performance?

- A: Same: Query performance stays about the same
- B: Not enough information: Definite answer cannot be given
- C: Slower: Query takes more time
- D: Faster: Query take less time

```
CREATE INDEX tbl_idx ON tbl (a, date_column);
```

```
SELECT date_column, count(*)  
FROM tbl  
WHERE a = 123  
GROUP BY date_column;
```

```
SELECT date_column, count(*)  
FROM tbl  
WHERE a = 123 AND b = 1  
GROUP BY date_column;
```