Databases II
Indexes and performance

HO
GENT

Contents

HO
GENT

Contents

HO
GENT

Is performance still relevant?

Because:

Transistor density on a manufactured semiconductor doubles about every 18 months.

Moore's law

(no longer valid since 2016?)

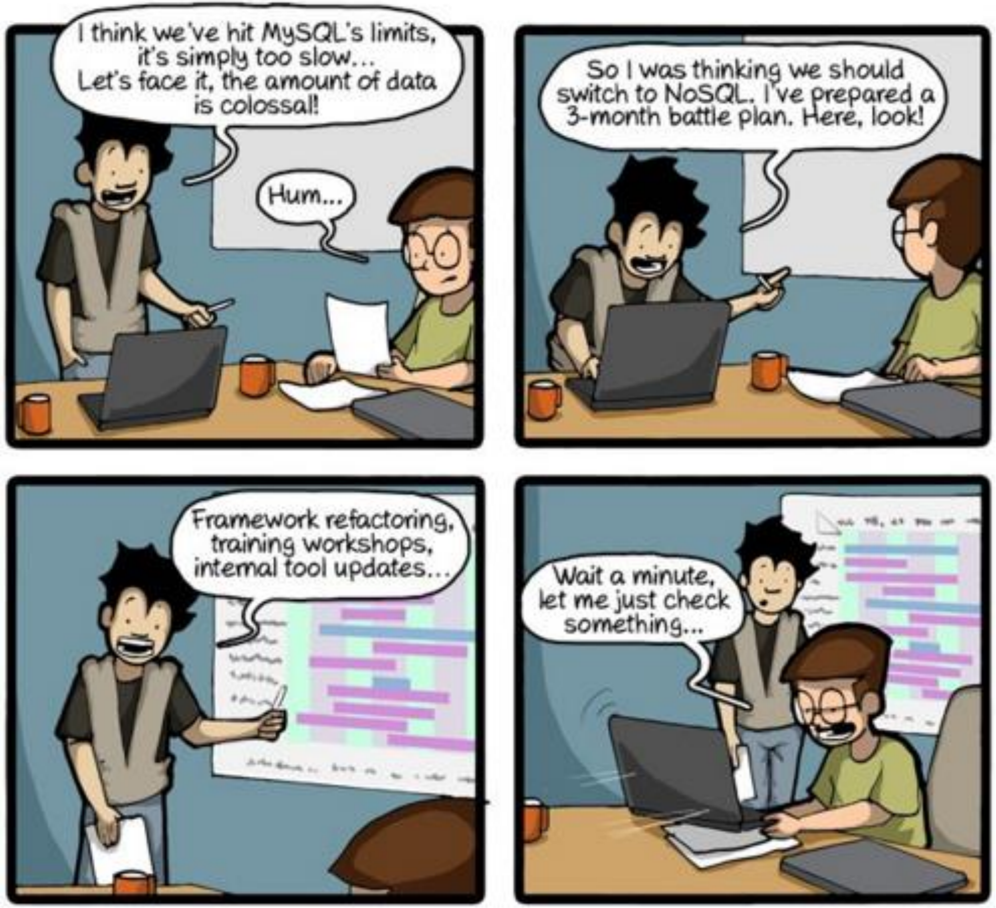
But:

Software gets slower faster than hardware gets faster

Wirth's law

HO
GENT

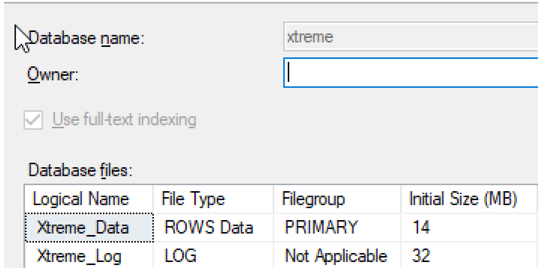
Anyway…

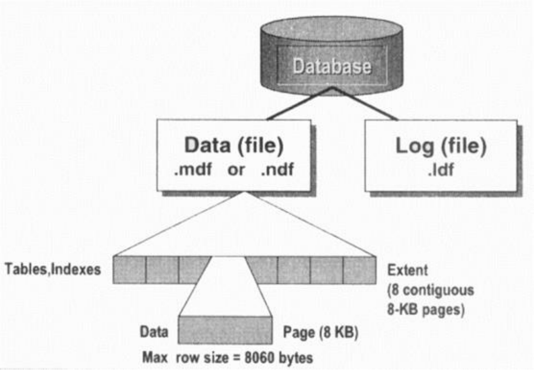

I think we've hit MySQL's limits, it's simply too slow... Let's face it, the amount of data is colossal!
Hum...
So I was thinking we should switch to NoSQL. I've prepared a 3-month battle plan. Here, look!
Framework refactoring, training workshops, internal tool updates...
Wait a minute, let me just check something...

Often indexes offer the solution

HO
GENT

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

HO
GENT

Contents

HO GENT

Clustered vs. Non-clustered indexes

- See the short version (2min) https://www.youtube.com/watch?v=AINh6_LqnDM
- See the long version (6min) https://www.youtube.com/watch?v=ITcOiLSfVJQ

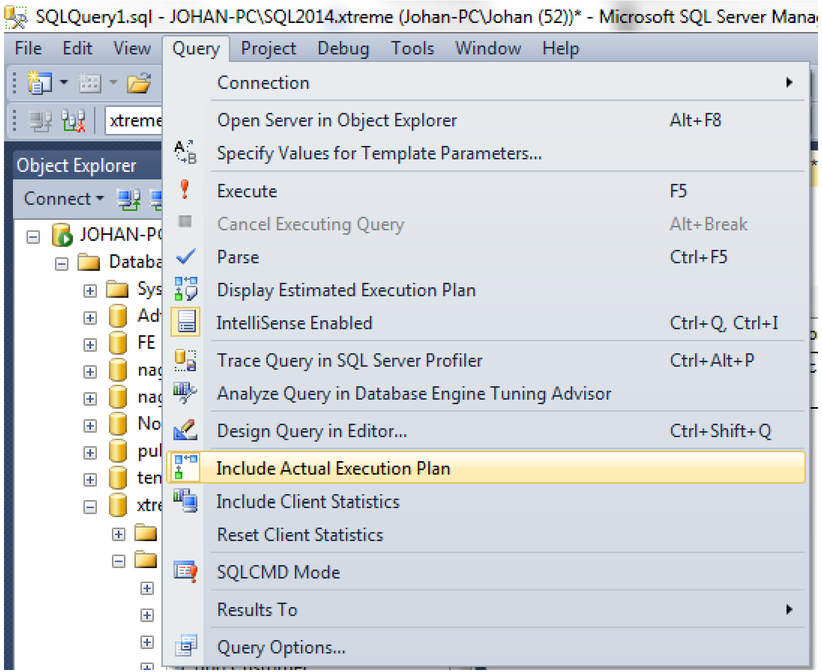
Table scan

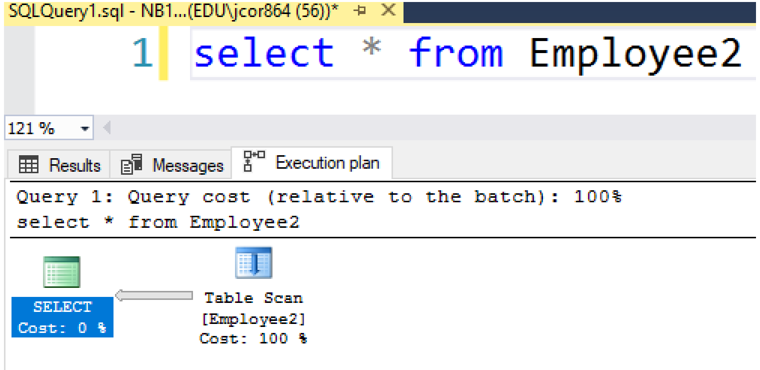
- Heap: unordered collection of data-pages without clustered index (see below) = default storage of a table
- Access via Index Alllocation Map (IAM)
- Table scan: if a query fetches all pages of the table → **always to avoid**!
- Other performance issues with heap:
  - Fragmentation: table is scattered 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

HO
GENT

Does my query cause a table scan?

Examine the Execution Plan of the query (DB xtreme + script EmployeeIdx)



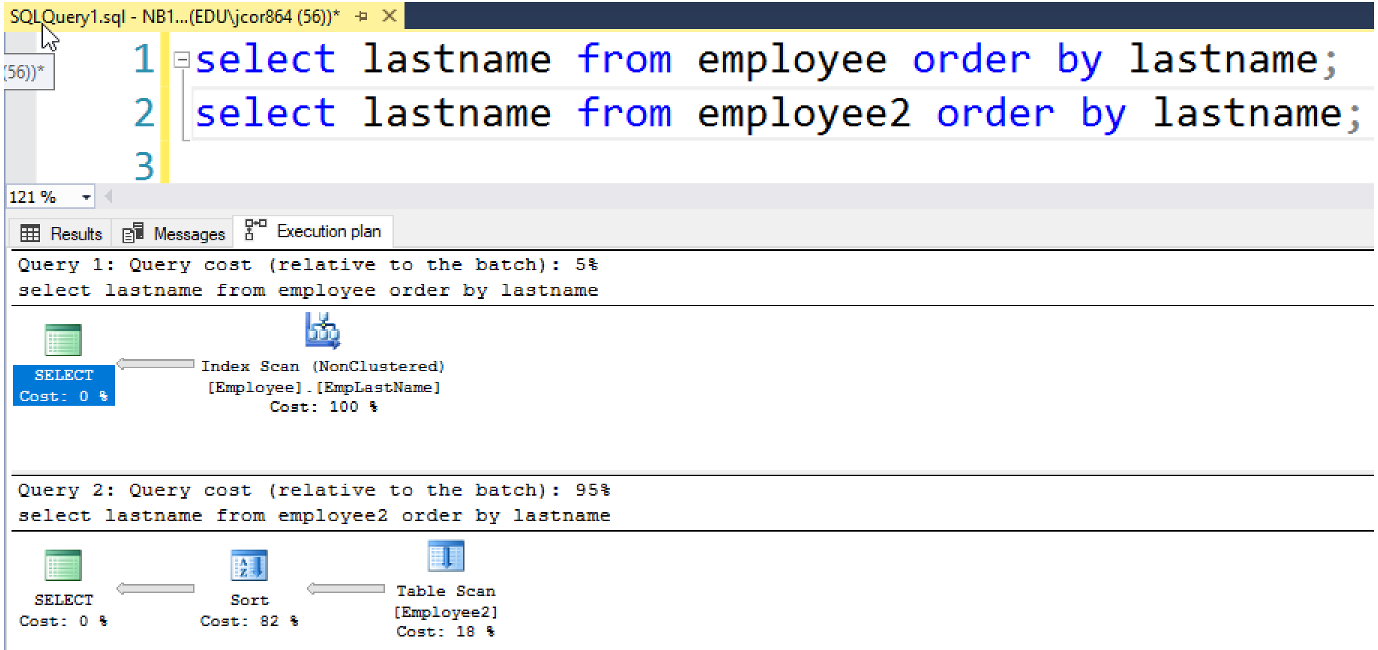


HO
GENT

Compare 2 queries

(DB xtreme + script EmployeeIdx)

- Execute the 2 queries together (select both + Execute!)
- Table Employee2 is a copy of Employee, but without indexes
- Query on Employee2 takes 19x longer!



HO
GENT

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 has to be updated too

HO GENT

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*

HO
GENT

Statistics

A cost-based optimizer uses statistics about tables, columns, and indexes.

Most statistics are collected on the **column** level:

- the number of distinct values
- the smallest and largest values (data range)
- the number of NULL occurrences
- the column histogram (data distribution).

The most important statistical value for a **table** is its size (in rows and blocks).

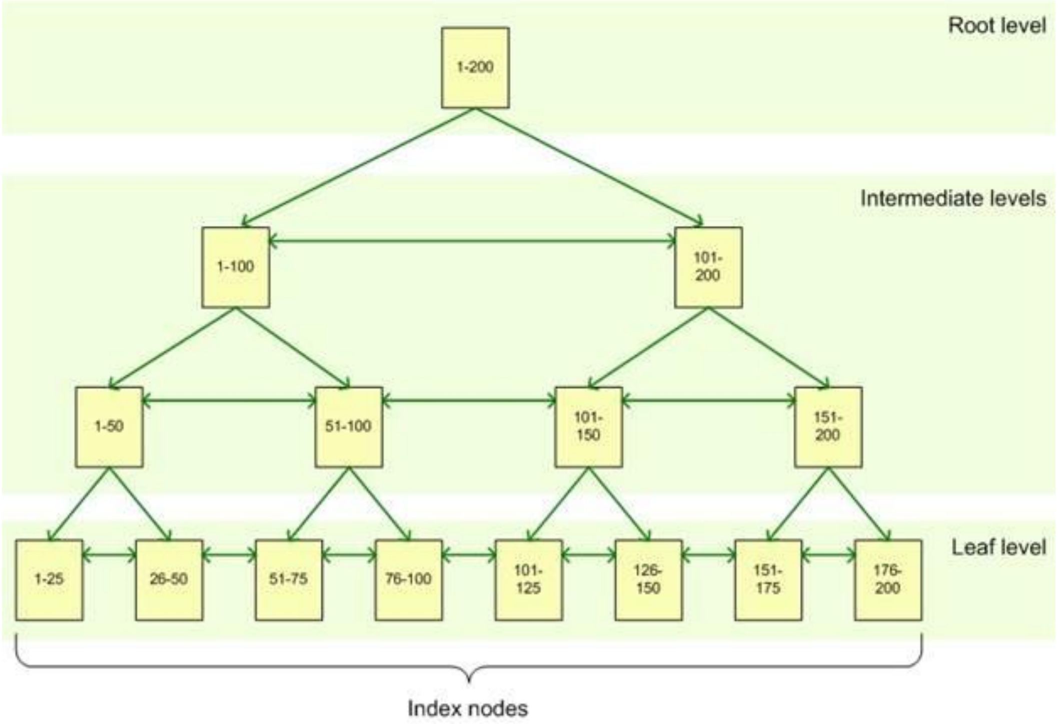
The most important **index** statistics are

- the tree depth
- the number of leaf nodes
- the number of distinct keys

The optimizer uses these values to estimate the selectivity of the **where** clause predicates.

HO
GENT

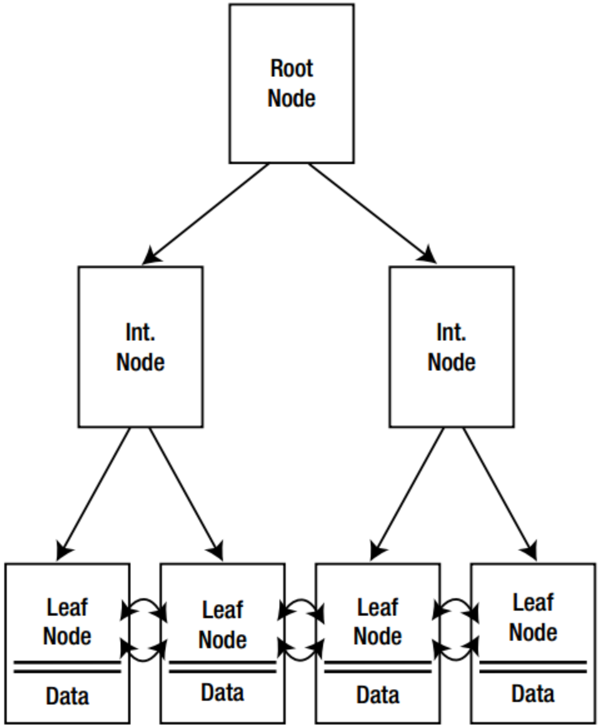
Indexes as B-trees



HO
GENT

Clustered index

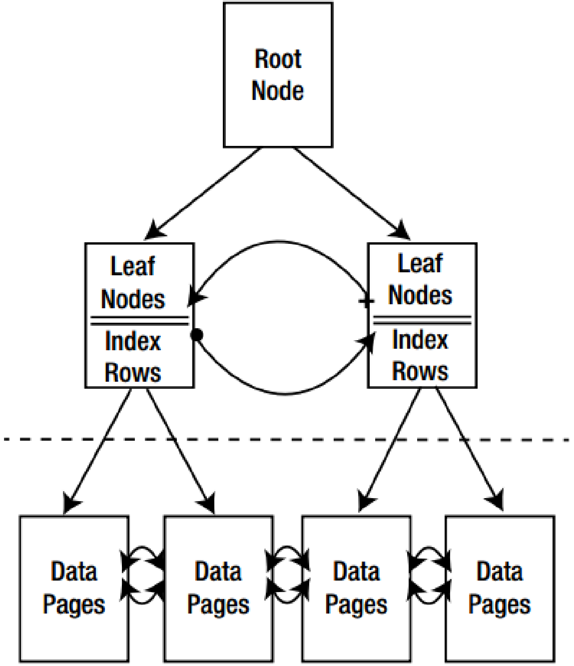
- The physical order of the rows in a table corresponds to the order in the clustered index.
- As a consequence, each table can have only one clustered index.
- The clustered index imposes unique values and the primary key constraint
- Advantages as opposed to table scan:
  - Double linked list ensures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



Int. Node = intermediate (tussenliggende) node

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 the query needs more fields than present in index, these fields have to be fetched from data pages.
- When reading via non-clustered index:

either:

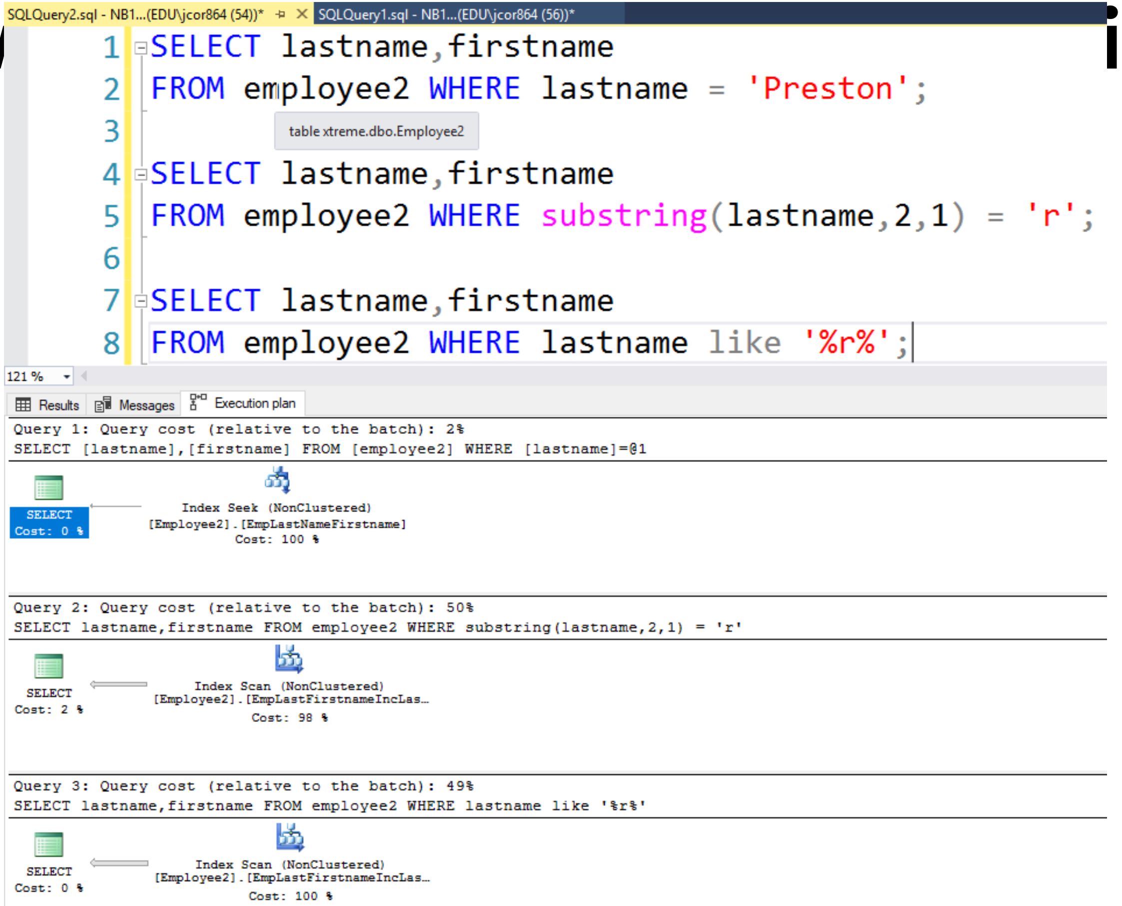
  - RID lookup = bookmark lookups to the heap using RID's (= row identifiers)

or:

  - Key lookup = bookmark lookups to a clustered index, if present

HO GENT

Use of indexes with functions and wildcards



HO
GENT

Use of indexes with functions and wildcards

- In the case of ‘%r%’ (with a leading wildcard) the index can’t be used for searching
- However, it may be advantageous to include the corresponding field in a covering index.
- That way, the data that is needed is “clustered” (= stored together), so fewer blocks have to be read.

HO
GENT

Contents

HO
GENT

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: example (DB xtreme, with script EmployeeIdx):

Current indexes on table Employee: each index indexes a single field.

- dbo.Employee
  - Columns
  - Keys
  - Constraints
  - Triggers
  - Indexes
    - EmpBirthdate (Non-Unique, Non-Clustered)
    - EmpFirstName (Non-Unique, Non-Clustered)
    - EmpLastName (Non-Unique, Non-Clustered)
    - EmpSalary (Non-Unique, Non-Clustered)
    - PK_Employee (Clustered)

Covering index: example (DB xtreme, with script EmployeeIdx):

SQLQuery1.sql - NB1...(EDU\jcor864 (56))*

```sql
select lastname from employee where lastname='Duffy';

select lastname,title from employee where lastname='Duffy';

```

121 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 33%
SELECT [lastname] FROM [employee] WHERE [lastname]=@1

SELECT        Index Seek (NonClustered)
Cost: 0 %     [Employee].[EmpLastName]
              Cost: 100 %
```

- Index seek via nonclustered index for seeking lastname = 'Duffy’

```
Query 2: Query cost (relative to the batch): 67%
SELECT [lastname],[title] FROM [employee] WHERE [lastname]=@1

SELECT        Nested Loops        Index Seek (NonClustered)
Cost: 0 %     (Inner Join)        [Employee].[EmpLastName]
              Cost: 0 %           Cost: 50 %

                                  Key Lookup (Clustered)
                                  [Employee].[PK_Employee]
                                  Cost: 50 %
```

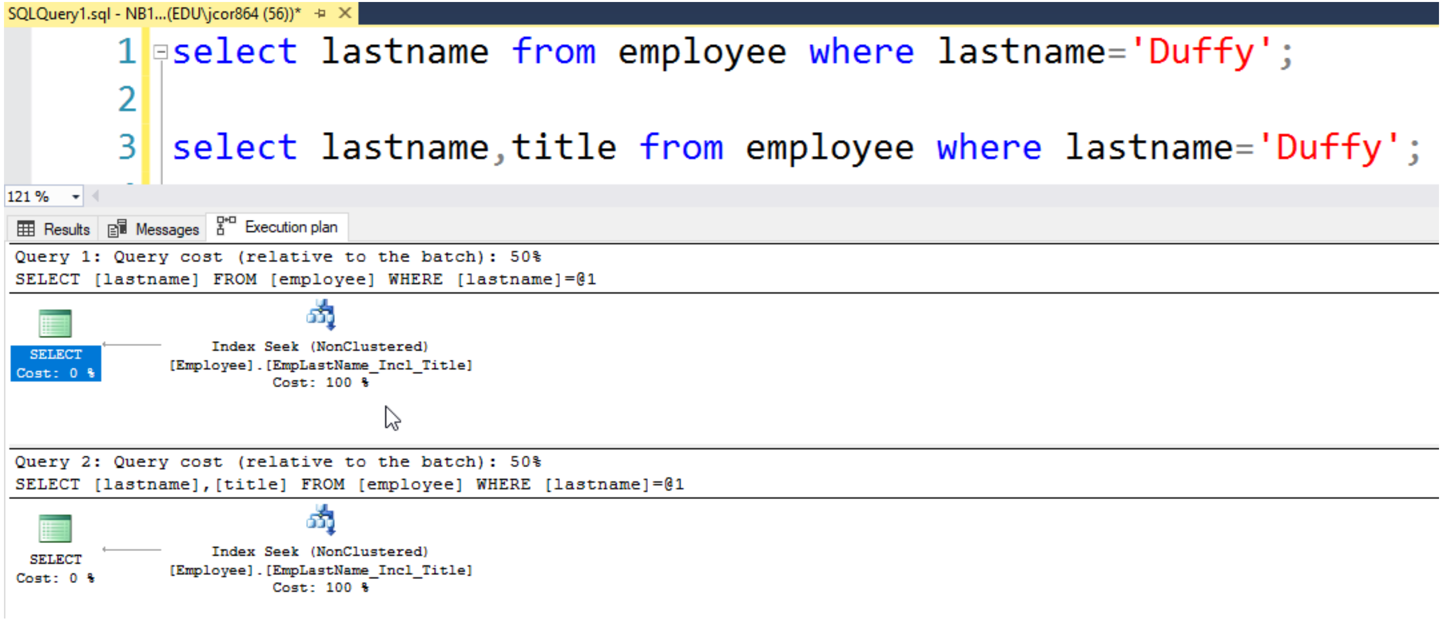
- Key look up in clustered index (= data) for fetching Title (not in index)

HO
GENT

Covering index: example (cont'd)

Solution: covering index via INCLUDE

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



HO
GENT

Contents

HO
GENT

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

```
create nonclustered index EmpLastName ON employee(lastname);
 +
create nonclustered index EmpFirstname ON employee(firstname);
```

OR ?

```
create nonclustered index EmpLastNameFirstname ON
employee(lastname, firstname);
```

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

Rule in SQL Server:

When querying (ex. in where-clause) only 2nd and or 3th, … field of index, it is not used. This directly follows from the B-tree table structure of the composed index

So:

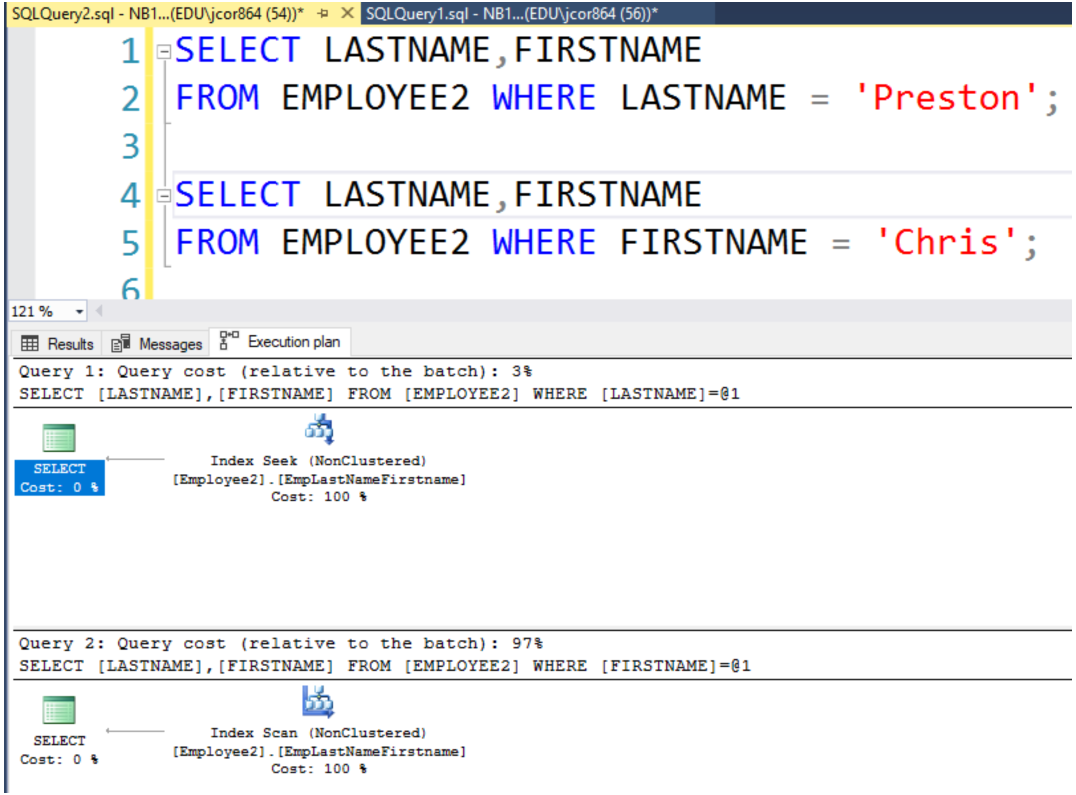
```
SELECT LASTNAME,FIRSTNAME
FROM EMPLOYEE2
WHERE FIRSTNAME = 'Chris';
```

does **not use** the double index.

Conclusion: make your indexes according to the most commonly used queries.

HO
GENT

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



Test:

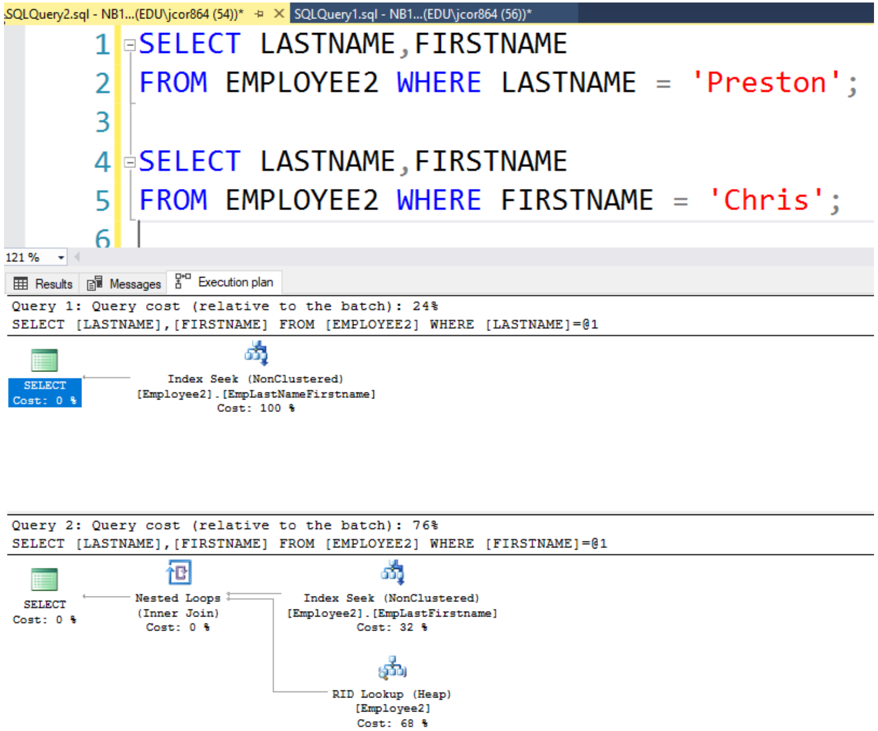
only combined index on

Lastname, Firstname

HO GENT

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

Extra index on Firstname: `create nonclustered index EmpFirstname ON employee2(firstname);`



not a spectacular improvement because of fetching lastname through lookup

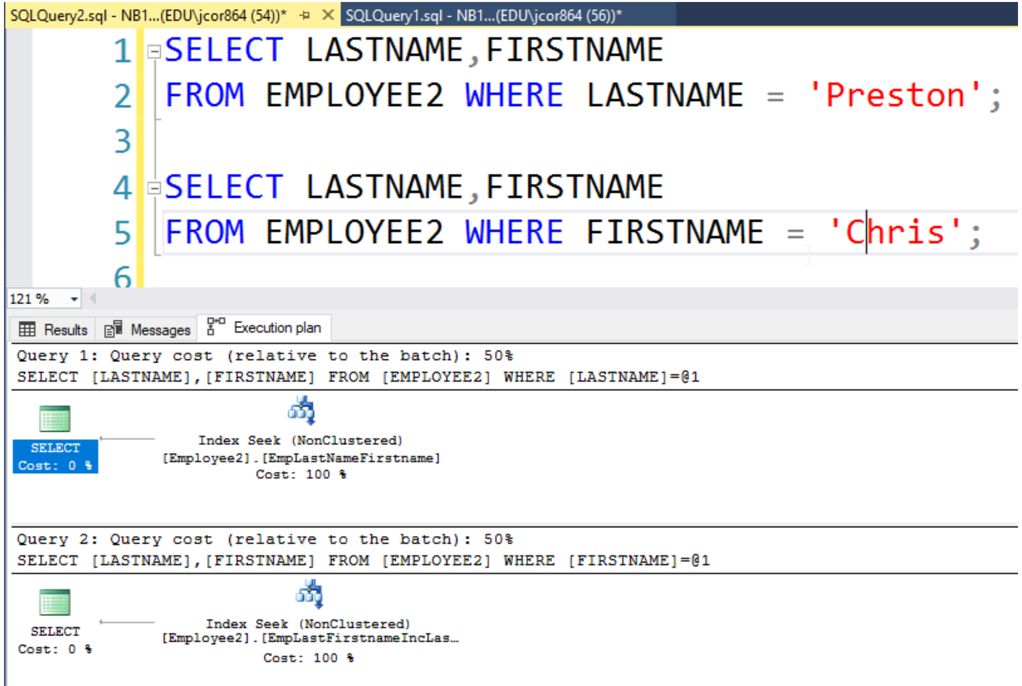
→ covering index with include 'lastname'

HO GENT

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

With extra index on Firstname and covering of Lastname

```
create nonclustered index EmpFirstnameIncLastname ON employee2(firstname)
                                                     INCLUDE (lastname);
```



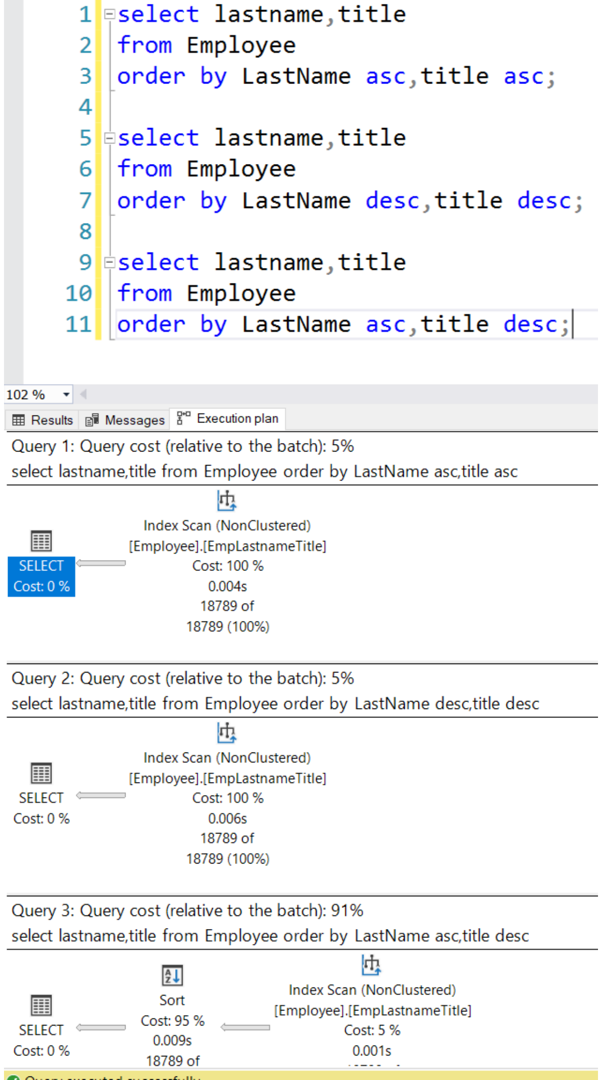
now query execution times are equal.

Indexes and performance

Sort order with concatenated indexes

```
CREATE NONCLUSTERED INDEX
EmpLastnameTitle ON Employee
(
      LastName ASC,
      Title ASC
)
```

Index can be used in reverse order, but you can't mix the order of the two fields.



Contents

HO
GENT

Creation of indexes: syntax

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

create index

```
create index ssnr_index on student(ssnr)
```

create 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.

Removing indexes

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

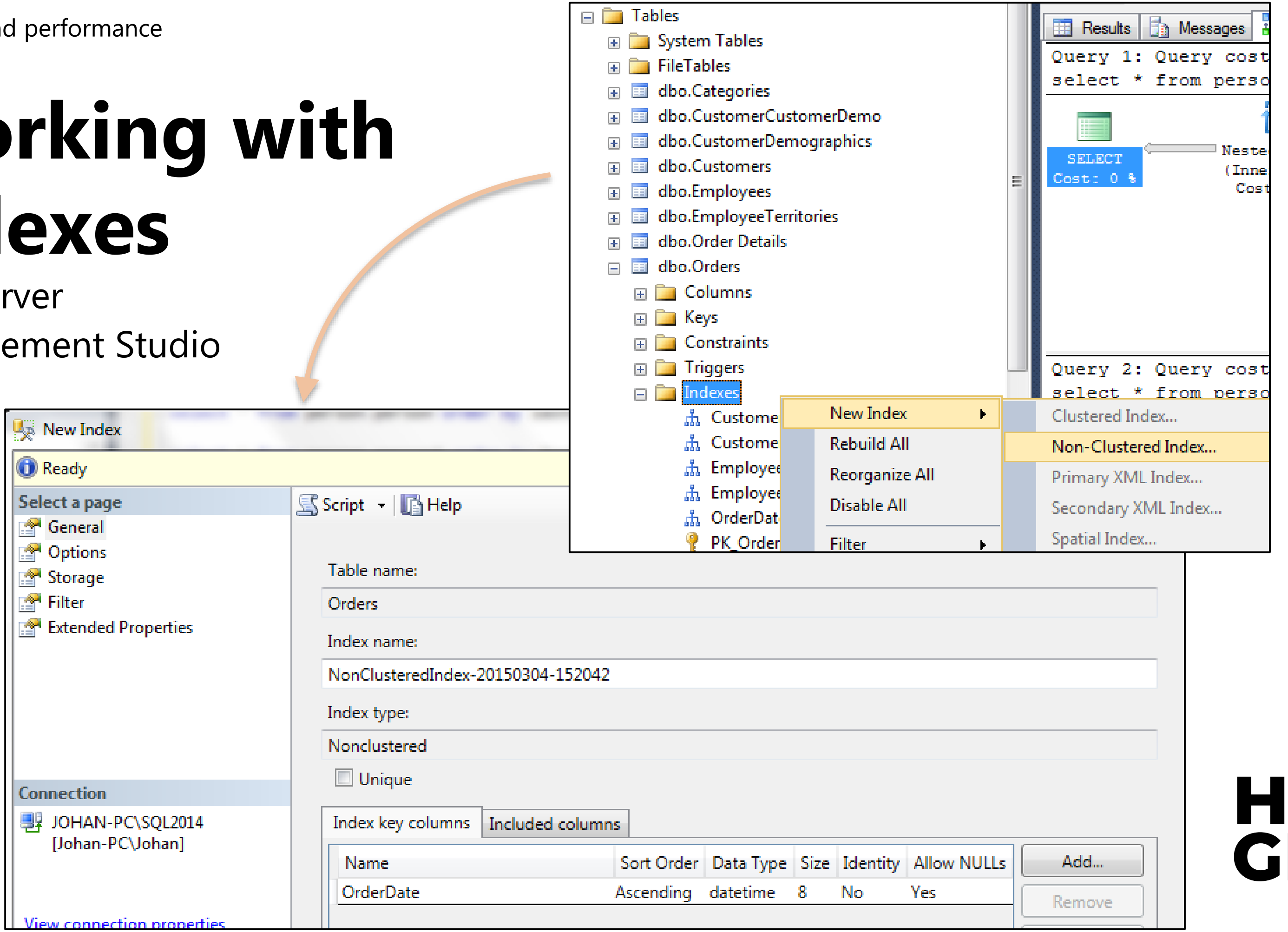
```
drop index student.SSNR_Index
```

deleting index

HO
GENT

Working with indexes

SQL Server
Management Studio



HO
GENT

Contents

HO GENT

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
- **Which columns should not be 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 of image

HO GENT

Rules of thumb

```sql
DB xtreme:

CREATE INDEX EmpFirstName ON
Employee (FirstName ASC);

CREATE INDEX EmpLastName ON Employee
(LastName ASC);

CREATE INDEX EmpDOB ON Employee
(BirthDate ASC);

CREATE INDEX EmpSalary ON Employee
(Salary ASC);
```

Employee

Column Name	Data Type	Allow Nulls
EmployeeID	int	☐
SupervisorID	int	☑
LastName	nvarchar(20)	☐
FirstName	nvarchar(10)	☐
Title	r(30)	☑
BirthDate	smalldatetime	☑
HireDate	smalldatetime	☑
HomePhone	nvarchar(20)	☑
Extension	nvarchar(4)	☑
Notes	ntext	☑
ReportsTo	int	☑
Salary	numeric(8, 2)	☑
SSN	nvarchar(12)	☑
Address	nvarchar(60)	☑
City	nvarchar(30)	☑
Region	nvarchar(30)	☑
Country	nvarchar(30)	☑
PostalCode	nvarchar(10)	☑
		☐

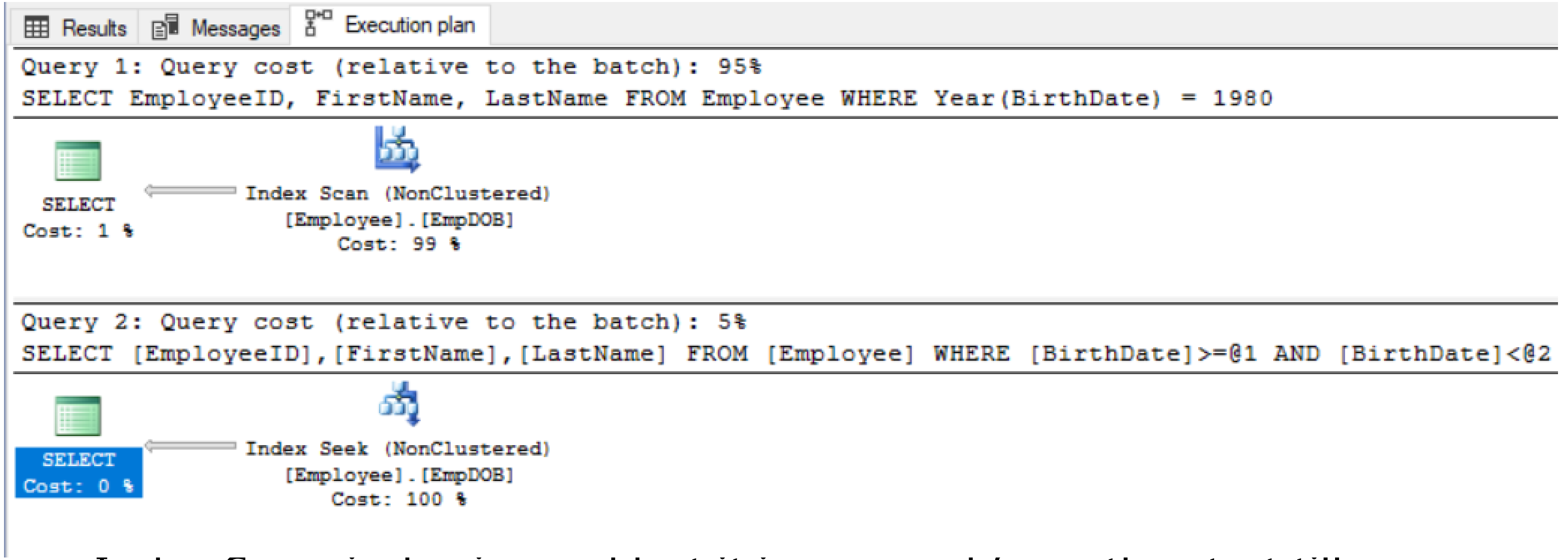
The following slides provides some general rules of thumb that are applicable in most cases on most databases. They are not carved in stone.
The employee table used in the examples has about 20.000 records.

(1) avoid the use of functions

```sql
-- BAD
SELECT EmployeeID, FirstName, LastName
FROM Employee
WHERE Year(BirthDate) = 1980;

-- GOOD
SELECT EmployeeID, FirstName, LastName
FROM Employee
WHERE BirthDate >= '1980-01-01'
AND BirthDate <  '1981-01-01';
```

(1) avoid the use of functions



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

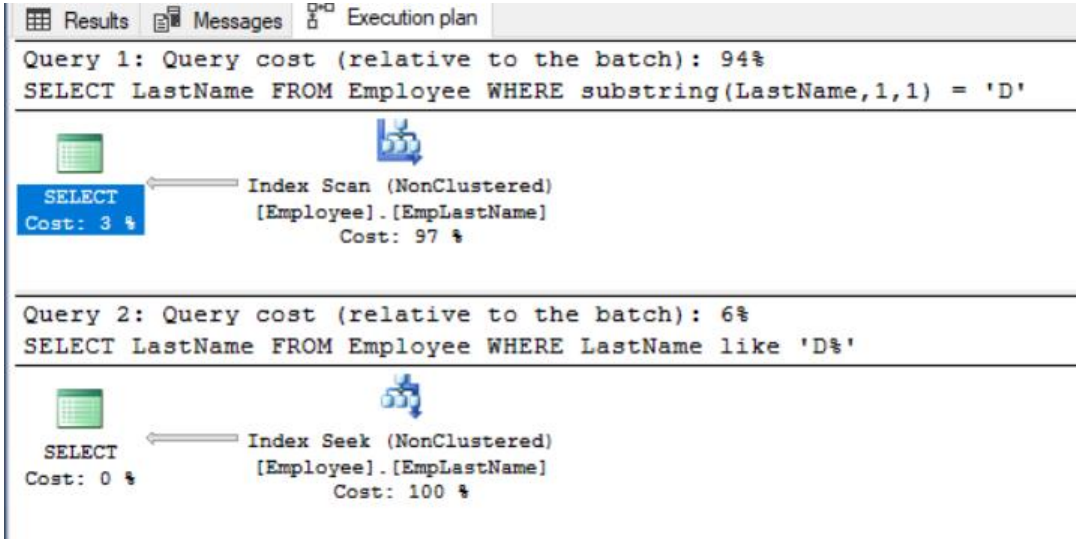
(1) avoid the use of functions

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

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

HO
GENT

(1) avoid the use of functions



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

(1) avoid the use of functions

- Somes DBMS's (like Oracle and PostgreSQL) support the creation of function based indexes
- SQL Server doesn't but it knows the concept of computed columns
- By creating an index on a computed column you simulate the effect of a function based index.

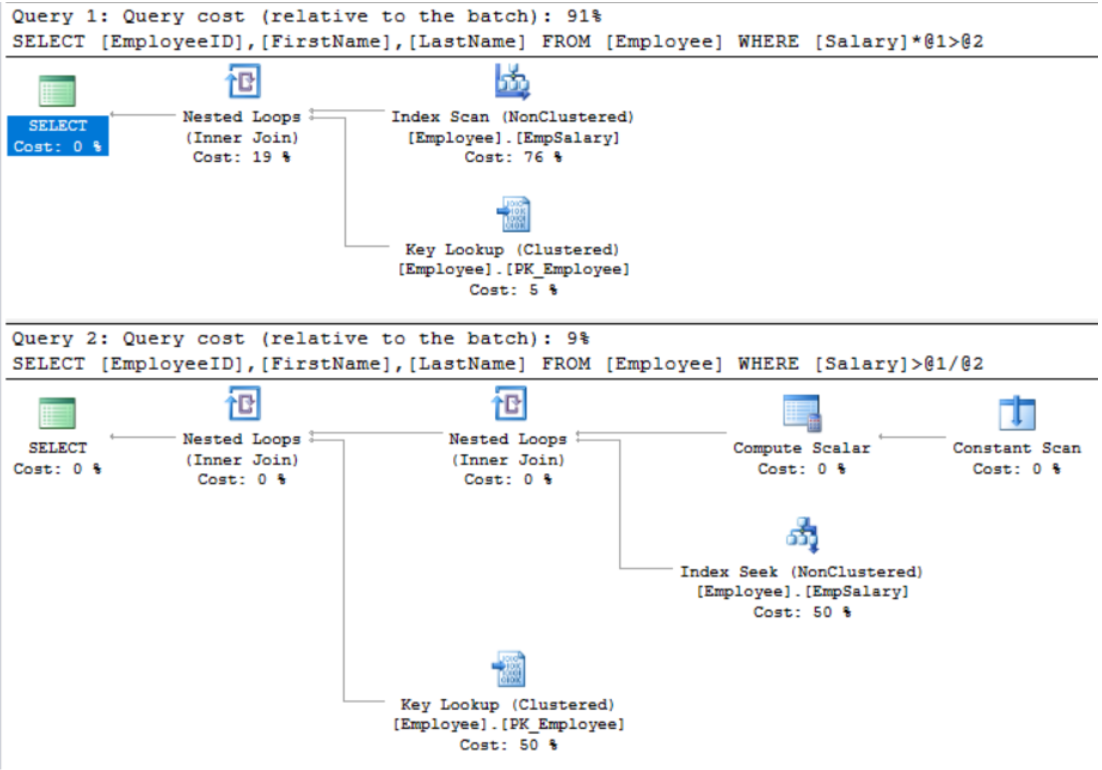
(2) avoid calculations, isolate columns

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

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

Rules of thumb

(2) avoid calculations, isolate columns



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”.

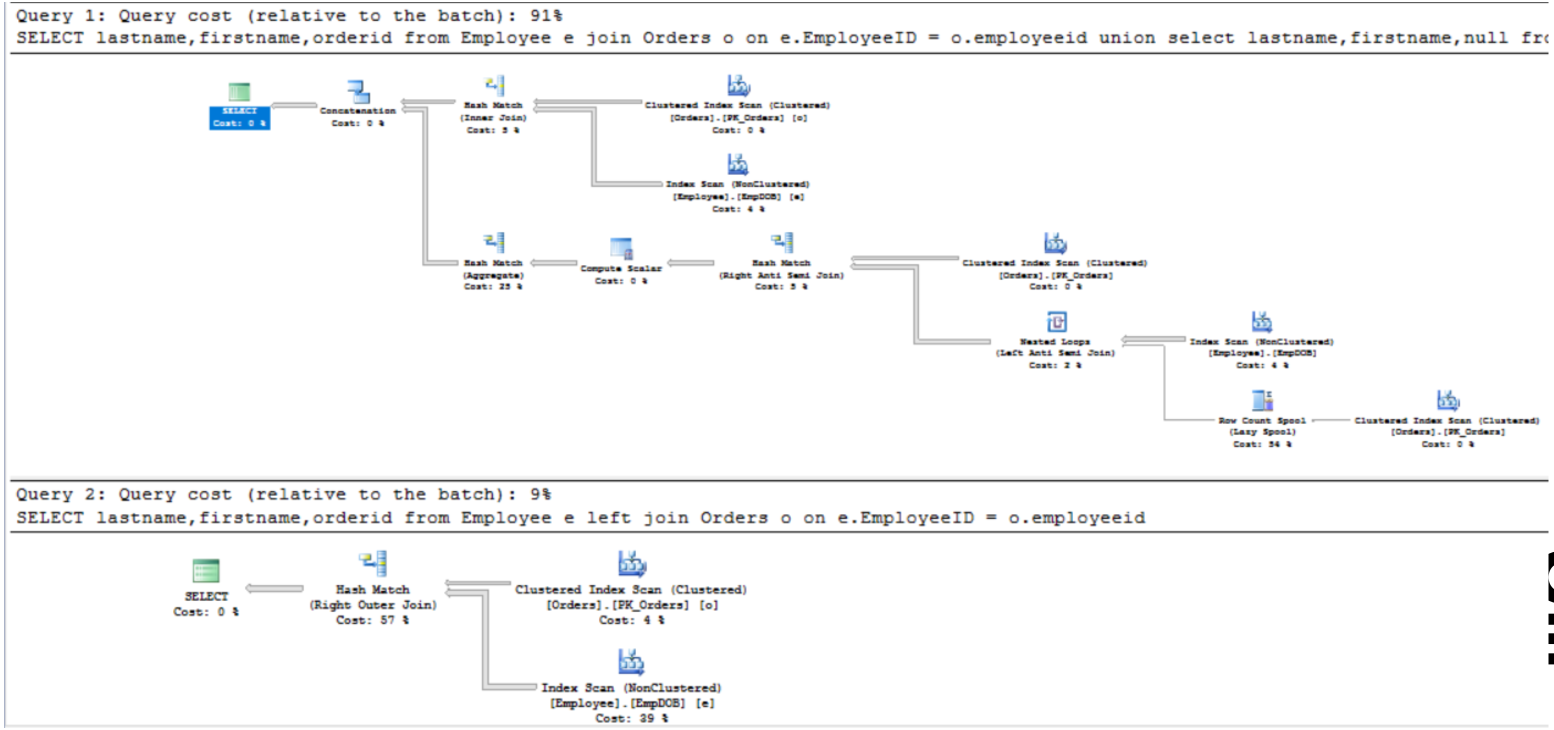
(2) prefer OUTER JOIN above UNION

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

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

HO
GENT

(3) prefer OUTER JOIN above UNION



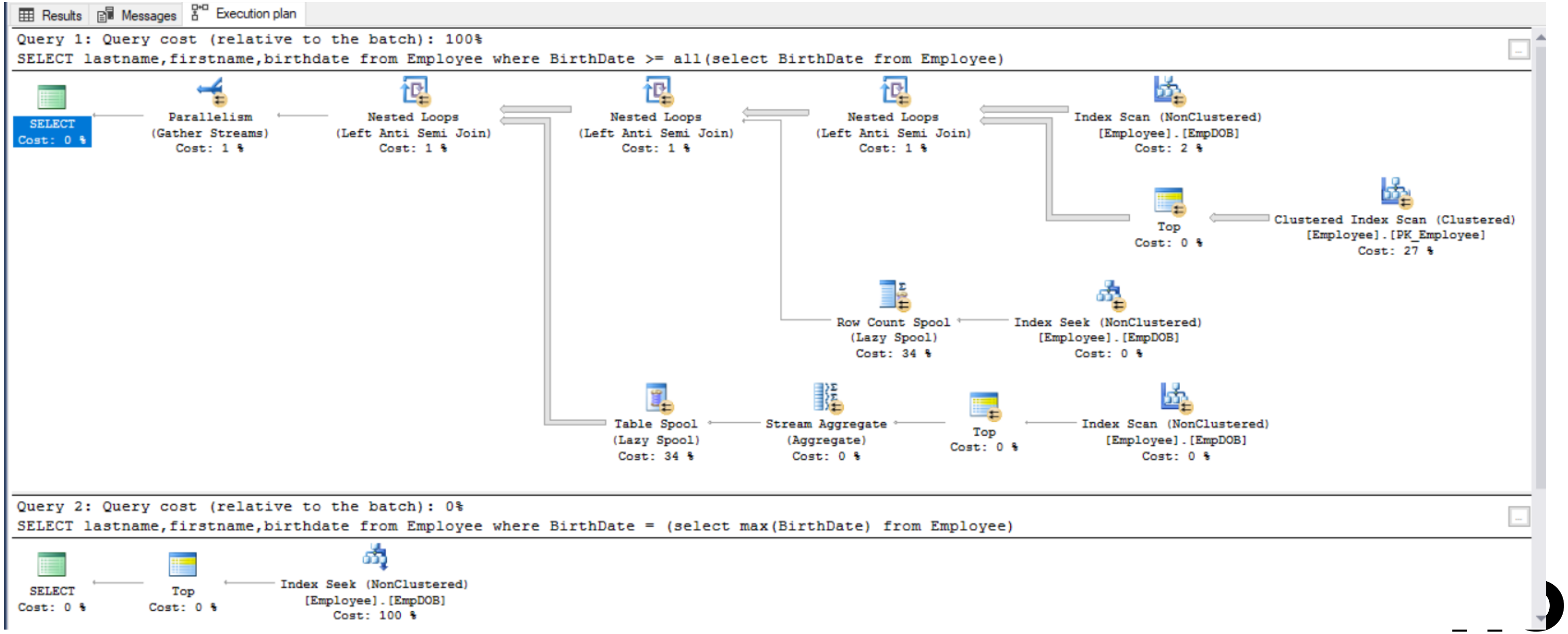
(4) avoid ANY and ALL

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

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

Rules of thumb

(4) avoid ANY and ALL



GENT

(5) Index for equality first — then for ranges.

```sql
select lastname, birthdate, country
from EmployeeHier
where BirthDate >= '1980-01-01' and BirthDate <= '1990-12-31'
and Country = 'Canada'

create index IdxCountryBirthdate on Employee(country,birthdate)
```

(6) Check the SQL code that is generated by your ORM tool or framework

- Get to know how your ORM tool generates SQL queries.

- e.g. Sometimes ORM tools use UPPER and LOWER without the developer’s knowledge. Hibernate, for example, injects an implicit LOWER for case-insensitive searches.

HO
GENT

(7) Avoid dynamic SQL whenever possible

```
declare @region varchar(10);
set @region = 'OR';
declare @sqlstring varchar(100) = 'select * from supplier
where region=''' + @region + '''';
exec (@sqlstring);
```

→ Disadvantages:

→ no cached query execution plan → slower

- debugging is more difficult (use PRINT!)
- Not allowed in UDF's (= risk of side-effect)
- SQL injection

HO GENT

(8) Use bind variables

- Bind parameters — also called dynamic parameters or bind variables — are an alternative way to pass data to the database.
- Instead of putting the values directly into the SQL statement, you just use a placeholder like ?, :name or @name and provide the actual values using a separate API call.
- Databases with an execution plan cache like SQL Server can reuse an execution plan when executing the same statement multiple times. It saves effort in rebuilding the execution plan but works only if the SQL statement is exactly the same.

(8) Use bind variables

C#

Without bind parameters:

```
int subsidiary_id;
SqlCommand cmd = new SqlCommand(
                   "select first_name, last_name"
                 + "  from employees"
                 + " where subsidiary_id = " + subsidiary_id
                 , connection);
```

Using a bind parameter:

```
int subsidiary_id;
SqlCommand cmd =
       new SqlCommand(
                      "select first_name, last_name"
                    + "  from employees"
                    + " where subsidiary_id = @subsidiary_id"
                    , connection);
cmd.Parameters.AddWithValue("@subsidiary_id", subsidiary_id);
```

HO GENT

(8) Use bind variables

JAVA

Without bind parameters:

```
int subsidiary_id;
Statement command = connection.createStatement(
                      "select first_name, last_name"
                    + "  from employees"
                    + " where subsidiary_id = " + subsidiary_id
                    );
```

Using a bind parameter:

```
int subsidiary_id;
PreparedStatement command = connection.prepareStatement(
                      "select first_name, last_name"
                    + "  from employees"
                    + " where subsidiary_id = ?"
                    );
command.setInt(1, subsidiary_id);
```

(8) Use bind variables

Cursor Sharing and Auto Parameterization

- The more complex the optimizer and the SQL query become, the more important execution plan caching becomes.
- The SQL Server and Oracle databases have features to automatically replace the literal values in a SQL string with bind parameters.
- These features are called CURSOR_SHARING (Oracle) or forced parameterization (SQL Server)
- These are workarounds for applications that do not use bind parameters at all.
- Enabling these features prevents developers from intentionally using literal values.

(8) Use bind variables

- Default in SQL Server: simple parameterization

  → optimizer will choose to use parameterization or not.
- Check by:
  SELECT name, is_parameterization_forced FROM sys.databases
- Can be turned into forced parameterization
- See
  - https://www.mssqltips.com/sqlservertip/2935/sql-server-simple-and-forced-parameterization/
  - https://docs.microsoft.com/en-us/sql/relational-databases/performance/specify-query-parameterization-behavior-by-using-plan-guides?view=sql-server-ver15

(9) Execute joins in the database.

- Don’t implement in your application what the database can do better
- Database is optimized for efficient data retrieval
- Limit network traffic

(10) Avoid unnecessary joins.

- Reading from many scattered tables is sensitive to disk seek latencies.
- JOIN can process only two tables at a time

Contents

HO
GENT

Quiz 1/5

Is the following index a good fit for the query?

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

```sql
SELECT * FROM tbl
WHERE YEAR(date_column) = 2017;
```

A. Good fit: No need to change anything
B. Bad fit: Changing the index or query could improve performance

HO GENT

Quiz 2/5

Is the following index a good fit for the query?

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

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

A. Good fit: No need to change anything

B. Bad fit: Changing the index or query could improve performance

HO
GENT

Quiz 3/5

Is the following index a good fit for both queries?

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

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

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

A. Good fit: No need to change anything

B. Bad fit: Changing the index or query could improve performance

Quiz 4/5

Is the following index a good fit for the query?

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

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

A. Good fit: No need to change anything

B. Bad fit: Changing the index or query could improve performance

HO
GENT

Quiz 5/5

This question is different.

First consider the following index and query:

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

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

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:

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

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

Contents

HO GENT

Index usage statistics

```
SELECT OBJECT_NAME(IX.OBJECT_ID) Table_Name
   ,IX.name AS Index_Name
   ,IX.type_desc Index_Type
   ,SUM(PS.[used_page_count]) * 8 IndexSizeKB
   ,IXUS.user_seeks AS NumOfSeeks
   ,IXUS.user_scans AS NumOfScans
   ,IXUS.user_lookups AS NumOfLookups
   ,IXUS.user_updates AS NumOfUpdates
   ,IXUS.last_user_seek AS LastSeek
   ,IXUS.last_user_scan AS LastScan
   ,IXUS.last_user_lookup AS LastLookup
   ,IXUS.last_user_update AS LastUpdate
FROM sys.indexes IX
INNER JOIN sys.dm_db_index_usage_stats IXUS ON IXUS.index_id = IX.index_id AND IXUS.OBJECT_ID = IX.OBJECT_ID
INNER JOIN sys.dm_db_partition_stats PS on PS.object_id=IX.object_id
WHERE OBJECTPROPERTY(IX.OBJECT_ID,'IsUserTable') = 1
GROUP BY OBJECT_NAME(IX.OBJECT_ID) ,IX.name ,IX.type_desc ,IXUS.user_seeks ,IXUS.user_scans
,IXUS.user_lookups,IXUS.user_updates ,IXUS.last_user_seek ,IXUS.last_user_scan ,IXUS.last_user_lookup
,IXUS.last_user_update
```

https://www.sqlshack.com/gathering-sql-server-indexes-statistics-and-usage-information/

Index usage statistics

Table_Name	Index_Name	Index_Type	IndexSizeKB	NumOfSeeks	NumOfScans	NumOfLookups	NumOfUpdates	LastSeek	LastScan	LastLookup	LastUpdate
categorie	PK_Categorieën	CLUSTERED	16	1194	0	0	0	2020-10-22 00:01:55.287	NULL	NULL	NULL
factuur	PK_factuur	CLUSTERED	40	2	103	0	2	2020-10-15 17:45:00.670	2020-10-21 22:21:39.750	NULL	2020-10-15 17:45:00.670
klant	PK_klant	CLUSTERED	16	1612790	972	0	0	2020-10-22 00:01:55.283	2020-10-22 00:01:55.087	NULL	NULL
kost	PK_Bedrijfskosten	CLUSTERED	48	0	1592	0	0	NULL	2020-10-22 00:01:55.287	NULL	NULL
param	NULL	HEAP	16	0	44	0	16	NULL	2020-10-21 18:06:18.603	NULL	2020-10-21 18:06:18.603
project	IX_project	NONCLUSTERED	112	47	391	0	2	2020-10-20 10:53:07.650	2020-10-22 00:00:27.583	NULL	2020-10-16 15:10:45.563
project	NonClusteredIndex-20160428-231519	NONCLUSTERED	112	2	21	0	2	2020-10-21 18:02:57.003	2020-10-21 18:02:06.220	NULL	2020-10-16 15:10:45.563
project	PK_project	CLUSTERED	112	73	24447	2	4	2020-10-21 18:03:31.510	2020-10-22 00:01:55.287	2020-10-21 18:02:57.003	2020-10-16 15:10:45.563
timesheet	PK_timesheet_1	CLUSTERED	224	0	8	0	8	NULL	2020-10-21 18:30:01.473	NULL	2020-10-21 18:30:01.473
timesheet	PK_timesheet_1	CLUSTERED	224	45	45468	0	106	2020-10-20 10:53:07.650	2020-10-22 00:01:55.287	NULL	2020-10-20 10:53:07.650
timesheetbackup	NULL	HEAP	1496	0	0	0	16	NULL	NULL	NULL	2020-10-20 10:53:07.640

- Which tables need indexes, which indexes are seldom used?
- Statistics since last start of SQL Server service:
  – Seek: index seek
  – Scan: index scan
  – Update: updates of index
  – Loopup: key lookup from nonclustered index in clustered index

References

"Effective SQL, 61 specific ways to Write Better SQL", Viescas et. al., 2017, Pearson Education

"SQL Performance Explained", Markus Winand, 2019

"Pro T-SQL 2012 Programmer's guide, 3d edition", Natarjan et. al., 2012, Apress

SQL Server 2017 Query Performance Tuning, Grant Fritchey, 2018, Apress

HO
GENT