SQL Server
Advanced Performance

Contents

- Introduction
- Clustered & Non-clustered Indexes
- Covering Indexes
- Concatenated Indexes
- Working with indexes
- Rules of thumb
- Quiz
- Materialized views
- Index statistics
- Storage & partitions

Contents

- **Introduction**
- Clustered & Non-clustered Indexes
- Covering Indexes
- Concatenated Indexes
- Working with indexes
- Rules of thumb
- Quiz
- Materialized views
- Index statistics
- Storage & partitions

Is performance still relevant?

Because:

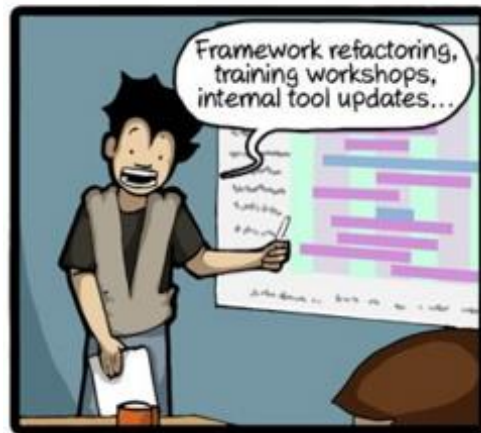*Transistor density on a manufactured semiconductor doubles about every 18 months.*

Moore's law

(no longer valid since 2016?)

But:

Software gets slower faster than hardware gets faster

Wirth's law

Anyway...

Often indexes offer the solution
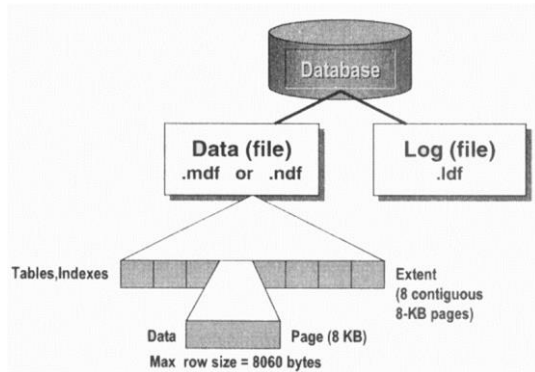
Space allocation by SQL Server

- 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

Contents

- Introduction
- **Clustered & Non-clustered Indexes**
- Covering Indexes
- Concatenated Indexes
- Working with indexes
- Rules of thumb
- Quiz
- Materialized views
- Index statistics
- Storage & partitions

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

Does my query cause a table scan?

Examine the Execution Plan of the query (db xtreme + script EmployeeIdx.sql)
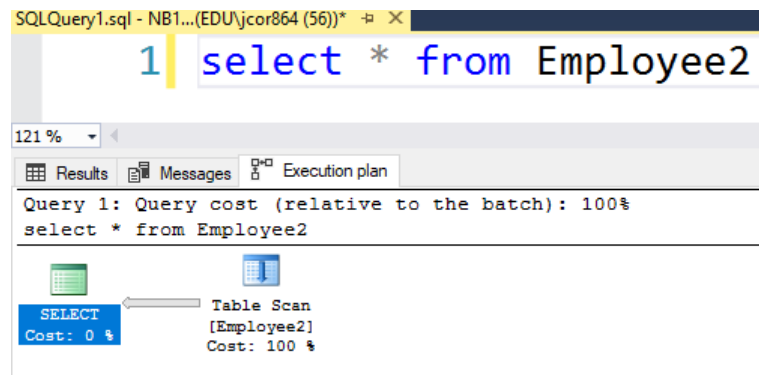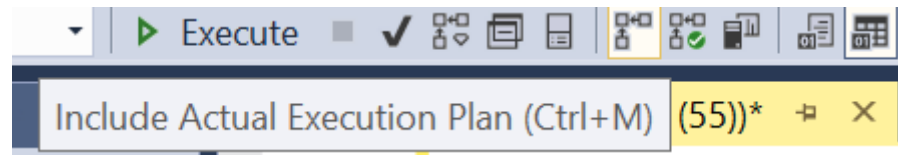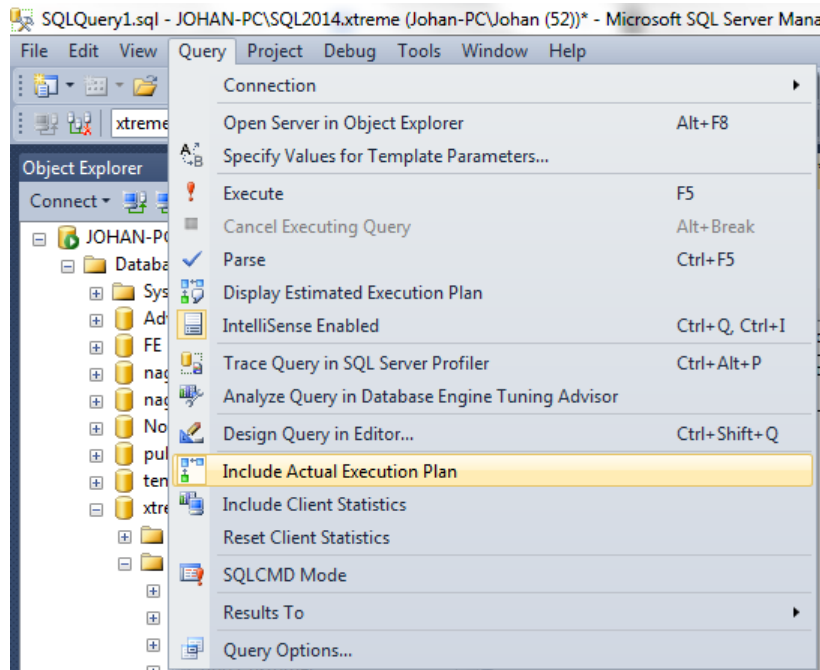
Compare 2 queries

(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 4x longer!

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

The query optimizer, or query planner, is the database component that transforms an SQL statement into an execution plan. This process is also called *compiling* or *parsing*. There are two distinct optimizer types.

- *Cost-based optimizers* (CBO) generate many execution plan variations and calculate a *cost* value for each plan. The cost calculation is based on the operations in use and the estimated row numbers. In the end the cost value serves as the benchmark for picking the "best" execution plan.
- *Rule-based optimizers* (RBO) generate the execution plan using a hardcoded rule set. Rule based optimizers are less flexible and are seldom used today.

The **execution plan is cached** so it can be reused when the same query arrives again before it is removed from the cache.

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.

Indexes as B-trees
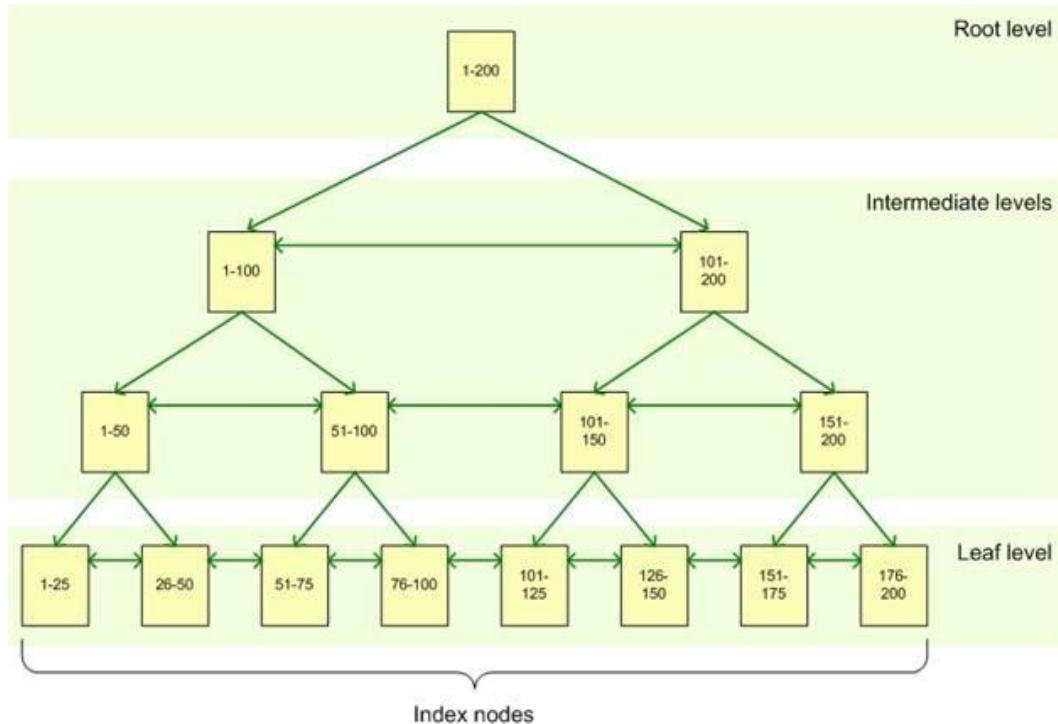
Clustered vs. Non-clustered indexes

- 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

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
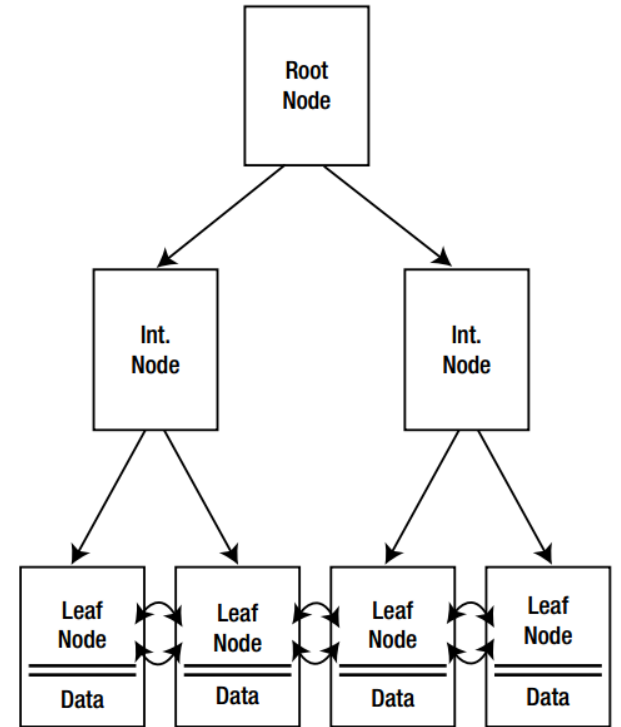
Int. Node = intermediate (tussenliggende) node

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
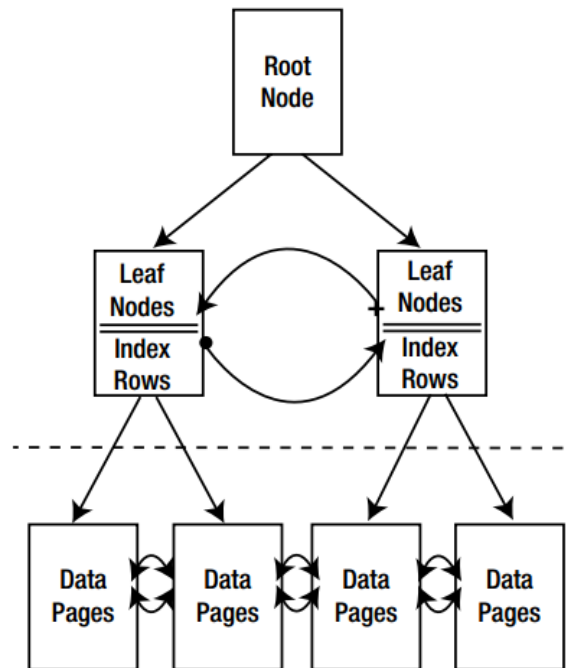
Non clustered index

- If 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

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.

Contents

- Introduction
- Clustered & Non-clustered Indexes
- **Covering Indexes**
- Concatenated Indexes
- Working with indexes
- Rules of thumb
- Quiz
- Materialized views
- Index statistics
- Storage & partitions

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.

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

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

```
1  select lastname from employee where lastname='Duffy';
2
3  select lastname title from employee where lastname='Duffy';
4
```

121 %

Results   Messages   Execution plan

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

SELECT
Cost: 0 %

Index Seek (NonClustered)
[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
Cost: 0 %

Nested Loops
(Inner Join)
Cost: 0 %

Index Seek (NonClustered)
[Employee].[EmpLastName]
Cost: 50 %

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

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

Covering index: example (cont'd)

Solution: covering index via INCLUDE

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

Use of indexes with functions and wildcards

Contents

- Introduction
- Clustered & Non-clustered Indexes
- Covering Indexes
- **Concatenated Indexes**
- Working with indexes
- Rules of thumb
- Quiz
- Materialized views
- Index statistics
- Storage & partitions

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.

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

Test:

only combined index on

Lastname, Firstname

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

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

```sql
SELECT LASTNAME,FIRSTNAME
FROM EMPLOYEE2 WHERE LASTNAME = 'Preston';

SELECT LASTNAME,FIRSTNAME
FROM EMPLOYEE2 WHERE FIRSTNAME = 'Chris';
```
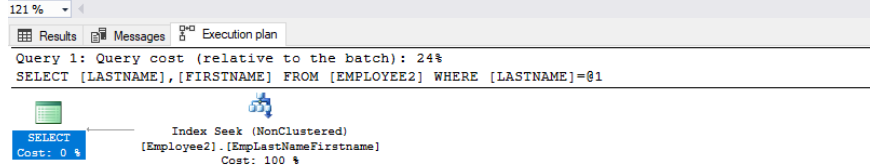
not a spectacular improvement because of fetching lastname through lookup

→ covering index with include 'lastname'

121 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 24%
SELECT [LASTNAME],[FIRSTNAME] FROM [EMPLOYEE2] WHERE [LASTNAME]=@1

SELECT              Index Seek (NonClustered)
Cost: 0 %          [Employee2].[EmpLastNameFirstname]
                    Cost: 100 %
```

```
Query 2: Query cost (relative to the batch): 76%
SELECT [LASTNAME],[FIRSTNAME] FROM [EMPLOYEE2] WHERE [FIRSTNAME]=@1

SELECT      Nested Loops      Index Seek (NonClustered)
Cost: 0 %   (Inner Join)      [Employee2].[EmpLastFirstname]
            Cost: 0 %          Cost: 32 %

                              RID Lookup (Heap)
                              [Employee2]
                              Cost: 68 %
```

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

With extra index on Firstname and covering of Lastname

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


now query execution times are equal.

Indexes and performance

Sort order with concatenated indexes

```sql
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.


```sql
1  select lastname,title
2  from Employee
3  order by LastName asc,title asc;
4
5  select lastname,title
6  from Employee
7  order by LastName desc,title desc;
8
9  select lastname,title
10 from Employee
11 order by LastName asc,title desc;
```

102 %

Results   Messages   Execution plan

Query 1: Query cost (relative to the batch): 5%
select lastname,title from Employee order by LastName asc,title asc

Index Scan (NonClustered)
[Employee].[EmpLastnameTitle]
SELECT   Cost: 100 %
Cost: 0 %   0.004s
                  18789 of
                  18789 (100%)

Query 2: Query cost (relative to the batch): 5%
select lastname,title from Employee order by LastName desc,title desc

Index Scan (NonClustered)
[Employee].[EmpLastnameTitle]
SELECT   Cost: 100 %
Cost: 0 %   0.006s
                  18789 of
                  18789 (100%)

Query 3: Query cost (relative to the batch): 91%
select lastname,title from Employee order by LastName asc,title desc

SELECT   Sort   Index Scan (NonClustered)
Cost: 0 %   Cost: 95 %   [Employee].[EmpLastnameTitle]
             0.009s   Cost: 5 %
             18789 of   0.001s

Contents

- Introduction
- Clustered & Non-clustered Indexes
- Covering Indexes
- Concatenated Indexes
- **Working with indexes**
- Rules of thumb
- Quiz
- Materialized views
- Index statistics
- Storage & partitions

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

Working with indexes

SQL Server
Management Studio

Tables
System Tables
FileTables
dbo.Categories
dbo.CustomerCustomerDemo
dbo.CustomerDemographics
dbo.Customers
dbo.Employees
dbo.EmployeeTerritories
dbo.Order Details
dbo.Orders
Columns
Keys
Constraints
Triggers
Indexes
Custome
Custome
Employee
Employee
OrderDat
PK_Order

New Index
Rebuild All
Reorganize All
Disable All
Filter

Clustered Index...
Non-Clustered Index...
Primary XML Index...
Secondary XML Index...
Spatial Index...

Results   Messages
Query 1: Query cost
select * from perso

SELECT
Cost: 0 %

Neste
(Inne
Cost

Query 2: Query cost
select * from perso

New Index

Ready

Select a page
General
Options
Storage
Filter
Extended Properties

Script   Help

Table name:
Orders

Index name:
NonClusteredIndex-20150304-152042

Index type:
Nonclustered

☐ Unique

Connection
JOHAN-PC\SQL2014
[Johan-PC\Johan]

View connection properties

Index key columns   Included columns

Name	Sort Order	Data Type	Size	Identity	Allow NULLs	Add...
OrderDate	Ascending	datetime	8	No	Yes	Remove

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

Looking for "expensive" queries

Install the performance dashboard according to the guidelines in

http://blog.sqlauthority.com/2014/09/22/sql-server-ssms-performance-dashboard-installation-and-configuration/

Start via Database/Reports/Custom reports …

Queries with Highest Duration
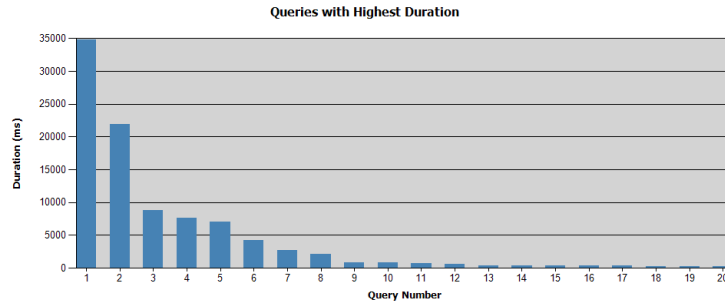
Example of a rapport: Expensive queries by duration:

Query Number	Representative Query	Total Executes	Cached Query Count	Unique Plan Count	Highest Plan Generation	Earliest Plan Cached	Cumulative CPU (ms) Total	Cumulative Duration (ms) Total	Max	Avg	Min	Cumulative Physical Reads Total	Cumulative Logical Reads Total
1	select * from person.person order by person.person.LastName.firstnam	6	3	1	1	3/4/2015 2:22:41 PM	4,378.468	34,813.075	24,848.507	5,802.179	1,874.773	3,926	415,90(
2	SELECT ISNULL((case dm.mirroring_redo_queue_type when N'UNLIMITED' then 0 else dm.mirroring_redo_queue end),0) AS [MirroringRedoQueueMaxSize], ISNULL(dm.mirroring_connection_timeout,0) AS [MirroringTimeout], ISNULL(dm.mirroring_partner_name,'') AS [MirroringPartner], ISNULL(dm.mirroring_partner_instance,'') AS [MirroringPartnerInstance], ISNULL(dm.mirroring_role,0) AS [MirroringRole], ISNULL(dm.mirroring_safety_level + 1, 0) AS [MirroringSafetyLevel], ISNULL(dm.mirroring_state + 1, 0) AS [MirroringStatus], ISNULL(dm.mirroring_witness_name,'') AS [MirroringWitness],	419	1	1	1	3/4/2015 11:51:51 AM	6,302.242	21,859.012	346.418	52.169	0.489	6,199	56,77(

Contents

- Introduction
- Clustered & Non-clustered Indexes
- Covering Indexes
- Concatenated Indexes
- Working with indexes
- **Rules of thumb**
- Quiz
- Materialized views
- Index statistics
- Storage & partitions

Rules of thumb

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);
```
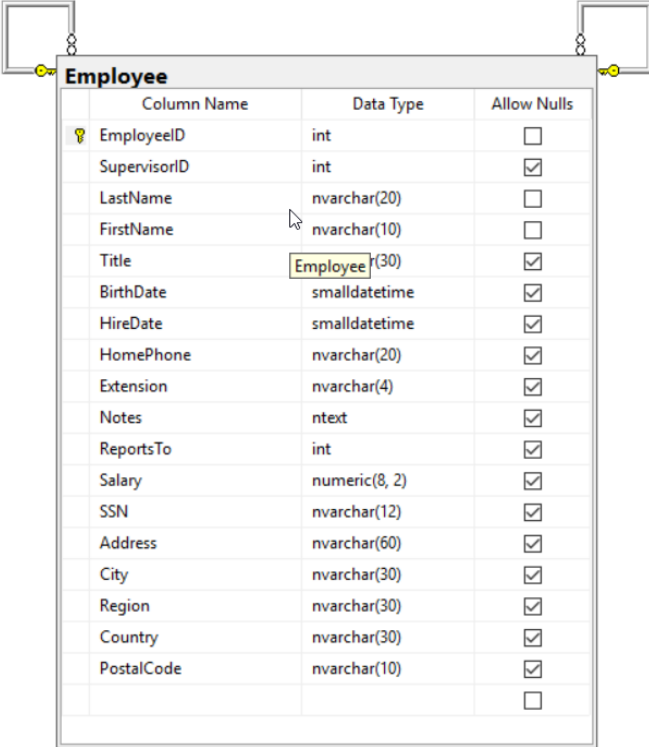
Employee		
Column Name	Data Type	Allow Nulls
🔑 EmployeeID	int	☐
SupervisorID	int	☑
LastName	nvarchar(20)	☐
FirstName	nvarchar(10)	☐
Title	Employee(30)	☑
BirthDate	smalldatetime	☑
HireDate	smalldatetime	☑
HomePhone	nvarchar(20)	☑
Extension	nvarchar(4)	☑
Notes	ntext	☑
ReportsTo	int	☑
Salary	numeric(8, 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


```
Results   Messages   Execution plan
Query 1: Query cost (relative to the batch): 95%
SELECT EmployeeID, FirstName, LastName FROM Employee WHERE Year(BirthDate) = 1980
```

```
SELECT              Index Scan (NonClustered)
Cost: 1 %              [Employee].[EmpDOB]
                         Cost: 99 %
```

```
Query 2: Query cost (relative to the batch): 5%
SELECT [EmployeeID],[FirstName],[LastName] FROM [Employee] WHERE [BirthDate]>=@1 AND [BirthDate]<@2
```

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

- <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

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

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

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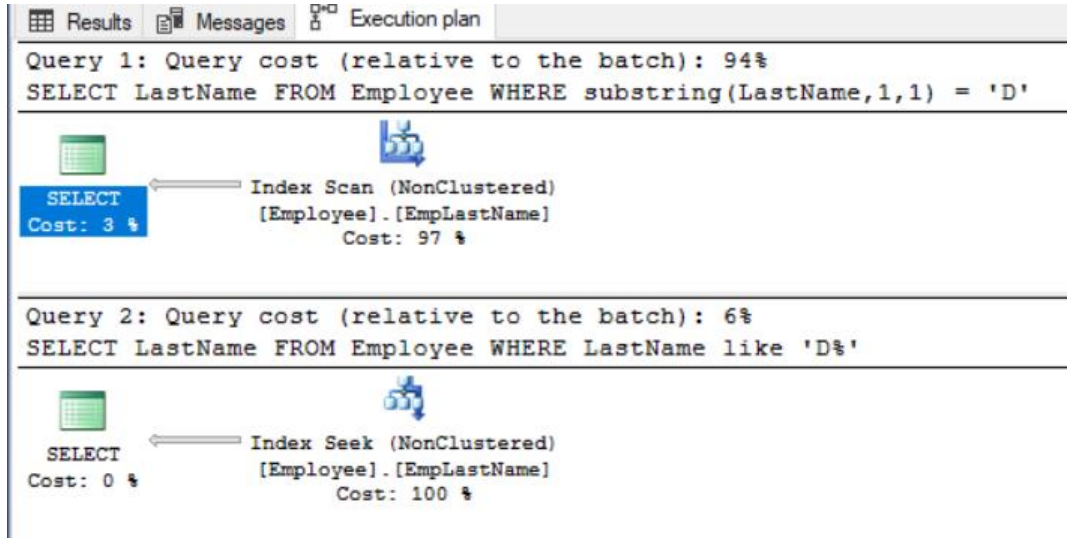
- <u>Index Scan</u>: index is used but it is scanned from the start till the searched records are found.
- <u>Index Seek</u>: tree structure of index is used, resulting in very fast data retrieval.

(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 concepts of computed columns and materialized views.
- By creating an index on a **computed column** or by creating a **materialized view** (see further) you can simulate the effect of a function based index.
- Computed column:

```
ALTER TABLE dbo.Product ADD RetailValue
    AS (Unitsinstock * UnitPrice * 1.5);

create index retailvalueindex on product(retailvalue)
include(productname,unitsinstock,price);

select productid,productname,unitsinstock,price,retailvalue
from Product order by RetailValue;
```

(1) avoid the use of functions

SQLQuery1.sql - NB2...(EDU\jcor864 (56))* ⊹ ✕

```
1  select productid,productname,unitsinstock*price
2  from product
3  order by price;
```

119 %

▦ Results   ▤ Messages   ▦ Execution plan

Query 1: Query cost (relative to the batch): 100%
select productid,productname,unitsinstock*price from product order by price

SELECT
Cost: 0 %

Compute Scalar
Cost: 0 %

Sort
Cost: 75 %
0.000s
117 of
117 (100%)

Clustered Index Scan (Clustered)
[Product].[PK_Product]
Cost: 25 %
0.000s
117 of
117 (100%)

- Expensive sort is done in query
- Can be avoided by using index.

(1) avoid the use of functions

```
 5   ALTER TABLE dbo.Product ADD RetailValue AS (Unitsinstock * Price * 1.5);
 6
 7   create index retailvalueindex on product(retailvalue) include(productname);
 8
 9   select productid,productname,retailvalue
10   from Product order by RetailValue;
11
```

119 %

Results   Messages   Execution plan

Query 1: Query cost (relative to the batch): 100%
select productid,productname,retailvalue from Product order by RetailValue

```
                                          Index Scan (NonClustered)
                                          [Product].[retailvalueindex]
  SELECT          Compute Scalar                Cost: 100 %
  Cost: 0 %         Cost: 0 %                     0.000s
                                                  117 of
                                                117 (100%)
```

- No more sort in query

- Sort is now through index, which is always sorted.
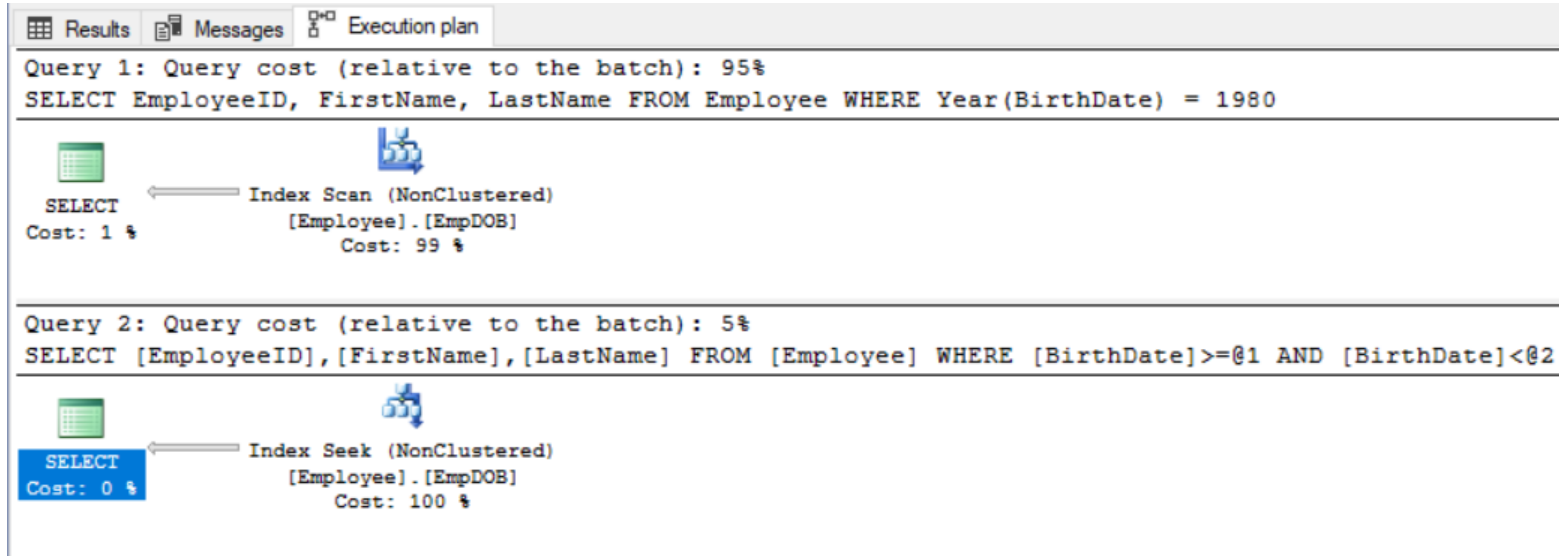
(2) avoid calculations, isolate columns

```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;
```

(2) avoid calculations, isolate columns


```
Query 1: Query cost (relative to the batch): 91%
SELECT [EmployeeID],[FirstName],[LastName] FROM [Employee] WHERE [Salary]*@1>@2
```

SELECT
Cost: 0 %

Nested Loops
(Inner Join)
Cost: 19 %

Index Scan (NonClustered)
[Employee].[EmpSalary]
Cost: 76 %

Key Lookup (Clustered)
[Employee].[PK_Employee]
Cost: 5 %

```
Query 2: Query cost (relative to the batch): 9%
SELECT [EmployeeID],[FirstName],[LastName] FROM [Employee] WHERE [Salary]>@1/@2
```

SELECT
Cost: 0 %

Nested Loops
(Inner Join)
Cost: 0 %

Nested Loops
(Inner Join)
Cost: 0 %

Compute Scalar
Cost: 0 %

Constant Scan
Cost: 0 %

Index Seek (NonClustered)
[Employee].[EmpSalary]
Cost: 50 %

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

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".

(3) prefer OUTER JOIN above UNION

```
-- 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;
```

(3) prefer OUTER JOIN above UNION

Query 1: Query cost (relative to the batch): 91%
SELECT lastname,firstname,orderid from Employee e join Orders o on e.EmployeeID = o.employeeid union select lastname,firstname,null fro
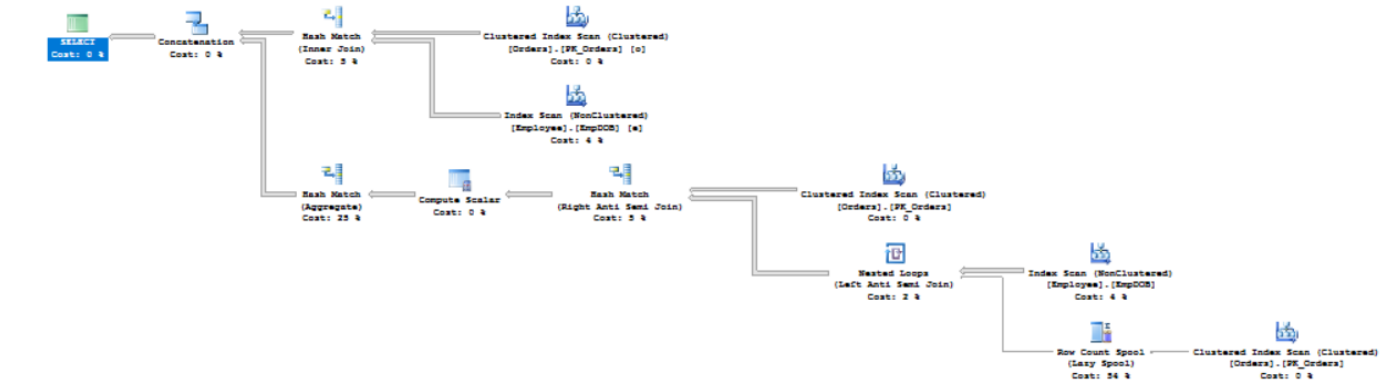
Query 2: Query cost (relative to the batch): 9%
SELECT lastname,firstname,orderid from Employee e left join Orders o on e.EmployeeID = o.employeeid
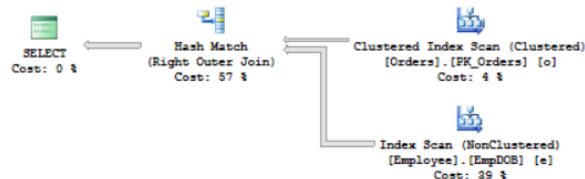
(4) avoid ANY and ALL

```
-- 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)
```

(4) avoid ANY and ALL
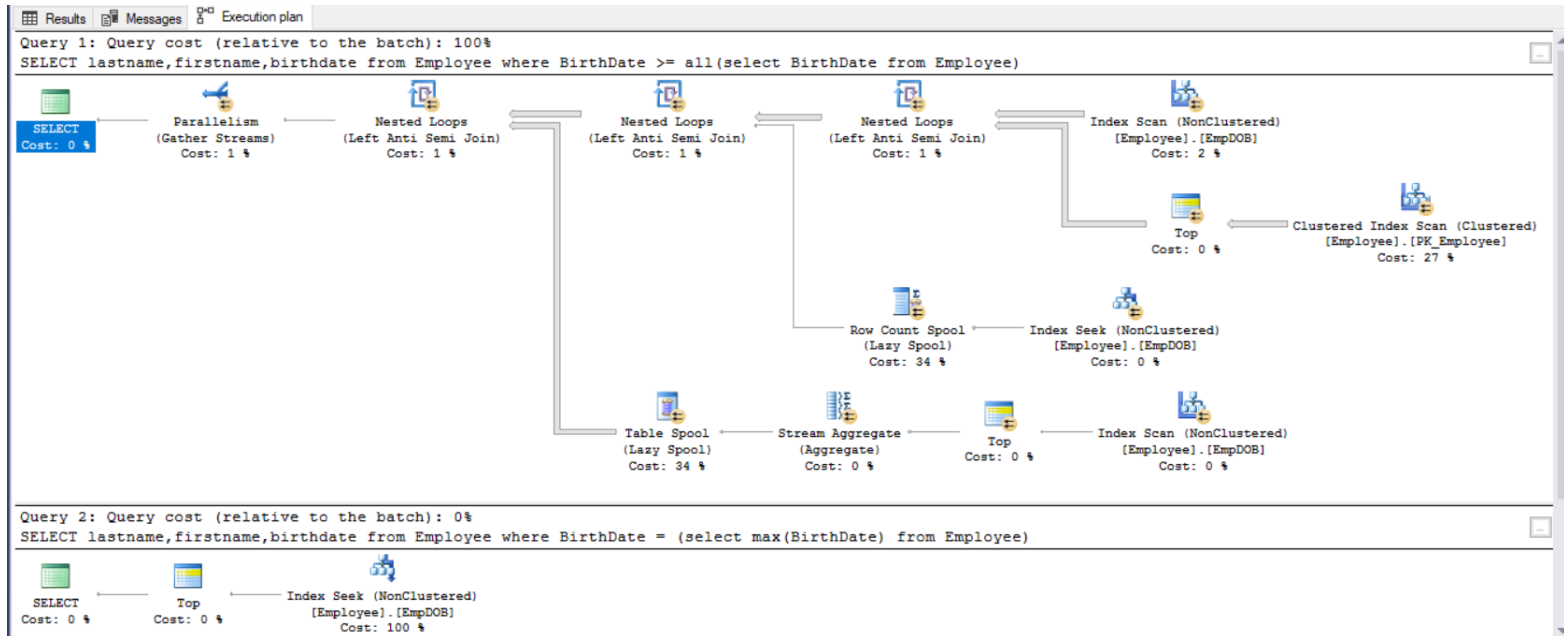
(5) Index for equality first — then for ranges.

```
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.

(7) Avoid dynamic SQL whenever possible

Dynamic SQL always involves late binding
- Late binding performs the binding of SQL-statements at runtime
  - additional flexibility is offered ("dynamic SQL")
  - syntax errors or authorisation issues will remain hidden until the program is executed
  - testing the application can be harder (use "PRINT")
  - less efficient for queries that must be executed multiple times
  - Risk of SQL injection attacks
- Not allowed in User Defined Functions in MS SQL Server (= risk of side effects)

(7) Avoid dynamic SQL whenever possible

```
-- Dynamic SQL example in t-SQL:
declare @region varchar(10);
set @region = 'OR';
declare @sqlstring varchar(100) = 'select * from supplier
where region=''' + @region + '''';
exec (@sqlstring);
```

(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);
```

(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

- Introduction
- Clustered & Non-clustered Indexes
- Covering Indexes
- Concatenated Indexes
- Working with indexes
- Rules of thumb
- **Quiz**
- Materialized views
- Index statistics
- Storage & partitions

Quiz 1/5

Is the following index a good fit for the query?

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

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

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

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
(you may assume field text is of datatype varchar(50) ?

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

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

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;
```

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

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;
```

Contents

- Introduction
- Clustered & Non-clustered Indexes
- Covering Indexes
- Concatenated Indexes
- Working with indexes
- Rules of thumb
- Quiz
- **Materialized views**
- Index statistics
- Storage & partitions

Indexed views (or materialized views)

- Database view in SQL Server: SELECT statement is stored, no data stored.
- A view is NOT a snapshot of the data at the moment the view is created.
- Created using the CREATE VIEW statement.
- You can write queries against a view as if it were a table.
- When a view gets queried, the optimizer receives the full definition of the SELECT statement and uses that as the basis for optimizing the query against the view.
- Through the optimization process, some or all of the definition of the SELECT statement may be used to satisfy the query against the view. What degree of simplification occurs here is determined by a combination of the SELECT statement itself and the query against that SELECT statement.

Indexed views (or materialized views)

- A database view can be **materialized** on the disk by **creating a unique clustered index** on the view.

- Such a view is referred to as an *indexed view* or a *materialized view.*

- After a unique clustered index is created on the view, the view's result set is materialized immediately and persisted in physical storage in the database, saving the overhead of performing costly operations during query execution.

- After the view is materialized, multiple nonclustered indexes can be created on the indexed view.

- This turns a view (again, just a query) into a real table with defined storage.

Indexed views: benefits

- Aggregations can be precomputed and stored in the indexed view to minimize expensive computations during query execution.

- Tables can be prejoined, and the resulting data set can be materialized.

- Combinations of joins or aggregations can be materialized.

Indexed views: restrictions

- The first index on the view must be a unique clustered index. Nonclustered indexes on an indexed view can be created only after the unique clustered index is created.
- The view definition must be *deterministic*—that is, it is able to return only one possible result for a given query. Functions like GETDATE(), RAND(), … are non-deterministic and cannot be used.
- The indexed view must reference only base tables in the same database, not other views.
- The indexed view may contain float columns. However, such columns cannot be included in the clustered index key.
- The indexed view must be schema bound (see further) to the tables referred to in the view to prevent modifications of the table schema.
- There are several restrictions on the syntax of the view definition. (A list of the syntax limitations on the view definition is provided in SQL Server Books Online.). Examples:
  - outer joins are not allowed
  - distinct is not allowed
  - COUNT has to be replaced by COUNT_BIG
  - Only COUNT_BIG(*) is allowed, COUNT_BIG(<field name>) is not allowed

Indexed views: schema binding

- In SQL Server, a **schema** is a **logical collection of database objects such as tables, views, stored procedures, indexes, triggers, functions**.
- It can be thought of as a **container**, created by a database user. The database user who creates a schema is the schema owner. Default schema = dbo, but many schemas can be created in a single database.
- The indexed view must be schema bound to the tables referred to in the view to prevent modifications of the table schema (frequently a major problem).
- In SQL Server, views are not bound to the schema of the base tables by default. In such case we may change the schema of the base table at any time, regardless of the fact that the associated view may or may not work with the new schema. We can even drop the underlying table while keeping the associated view without any warning. In this case when the view is used, we will get an invalid object name error for the base table.
- If you want to create an index on a view or you want to preserve the base table schema once a view has been defined, in both these cases you have to use the "WITH SCHEMABINDING" clause to bind the view to the schema of the base tables (db xtreme):

```
CREATE VIEW dbo.V_ProductsCustomer(productcode, customername, sumquantity)
WITH SCHEMABINDING
AS SELECT productid, customername, sum(quantity)
    FROM dbo.customer
JOIN dbo.orders ON orders.customerid = customer.customerid
JOIN dbo.ordersdetail ON orders.orderid = ordersdetail.orderid
GROUP BY  productid,customername
```

Indexed views: schema binding

- Be aware that if you do not use the schema name, dbo in this case, then you will get the following error while creating the view. "Cannot schema bind view 'dbo.vw_sampleView' because name 'SAMPLETABLE' is invalid for schema binding. Names must be in two-part format and an object cannot reference itself."
- So the steps are:
    1. Create view with schema binding (use schema name).
    2. Create clustered index on view to materialize the view (physical storage = index)
    3. Create any non clustered indexed on view to speed up queries.

Indexed view: example

- Query to optimize (db xtreme)

```sql
select orderid,productid,unitprice*quantity linevalue
from ordersdetail
order by unitprice*quantity desc;
```
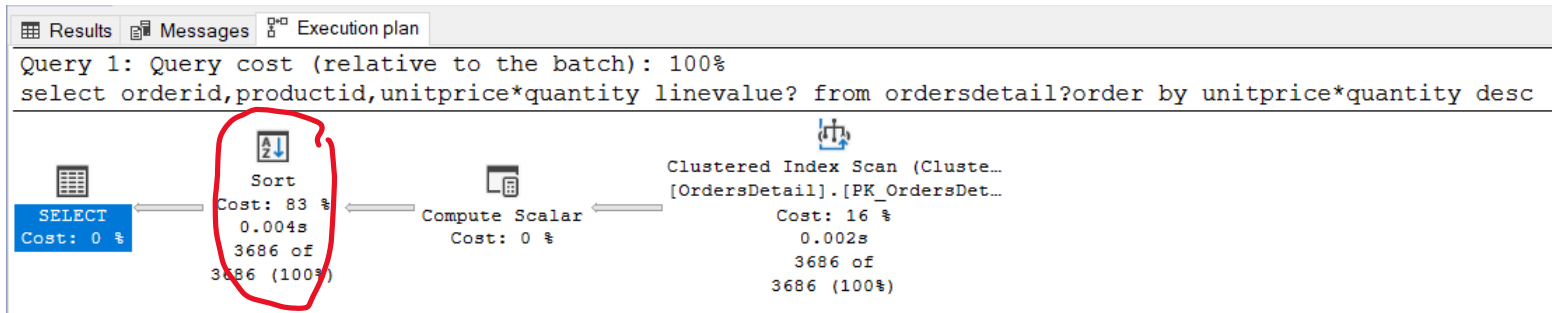

- Solution 1: calculated field (see above)

- Solution 2: indexed view

Indexed view: example (cont'd)

Create view

```
create view dbo.ordersdetaillinevalue(orderid,productid,linevalue)
with schemabinding
as select orderid,productid,unitprice*quantity
from dbo.ordersdetail
```

Create clustered index to create storage:

```
create unique clustered index detail_pk on ordersdetaillinevalue(orderid,productid)
```

Create index to speed up query:

```
create nonclustered index detail_idx on ordersdetaillinevalue(linevalue)
include(orderid,productid)
```

Check execution plan:
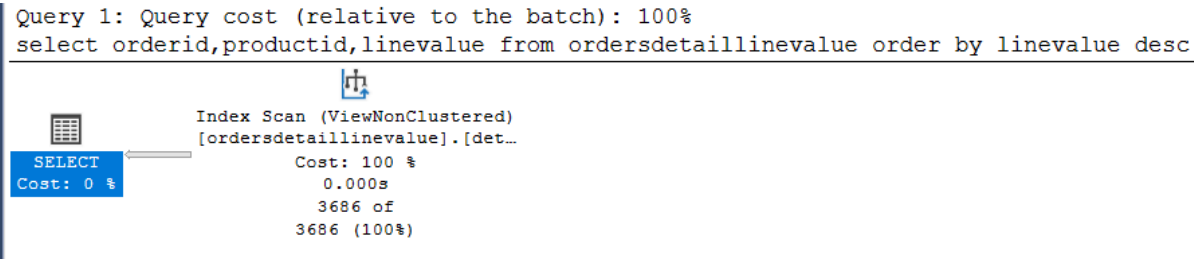

```
Query 1: Query cost (relative to the batch): 100%
select orderid,productid,linevalue from ordersdetaillinevalue order by linevalue desc
```

Index Scan (ViewNonClustered)
[ordersdetaillinevalue].[det…
Cost: 100 %
0.000s
3686 of
3686 (100%)

SELECT
Cost: 0 %

Contents

- Introduction
- Clustered & Non-clustered Indexes
- Covering Indexes
- Concatenated Indexes
- Working with indexes
- Rules of thumb
- Quiz
- Materialized views
- **Index statistics**
- Storage & partitions

Index usage statistics

Overview of indexes and their usage:

```sql
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
  - Lookup: key lookup from nonclustered index in clustered index

Contents

- Introduction
- Clustered & Non-clustered Indexes
- Covering Indexes
- Concatenated Indexes
- Working with indexes
- Rules of thumb
- Quiz
- Materialized views
- Index statistics
- **Storage & partitions**

Storage: files & filegroups

- Files & Filegroups
  - each DB consists of at least two files
    - Data file (.mdf)
    - log file (.ldf)
    - Optionally extra data files can be added (.ndf)
  - filegroup
    - Logic grouping of files for administration and allocation purposes
    - default: 1 filegroup per db-file
- Performance gain by organizing file groups
  - Store file groups on separate physical disks
  - Store data files and non clustered indexes on separate disks (clustered index IS data file)
  - Store log files on another disk than data and non clustered indexes for data security

Storage: files & filegroups

- Moving a table to another file group
  - Drop current clustered index
  - Remember: clustered index = data file
  - Create new clustered index and specify filegroup with CREATE CLUSTERED INDEX command.

Storage: space allocation

- SQL Server uses random access files
- Space allocation in *extents* en *pages*
- Page = 8 kB blok contiguous space
- Extent = 8 logical continguous pages.
  - uniform extents: for a single db object (table, index, …)
  - mixed extents: can be shared by 8 database objects
- Each new table or index is allocated in a mixed extent
- If table or index exceeds 8 pages: in uniform extent

Storage: example

```sql
CREATE TABLE dbo.SmallRows
(
Id int NOT NULL,
LastName nchar(50) NOT NULL,
FirstName nchar(50) NOT NULL,
MiddleName nchar(50) NULL
);

INSERT INTO dbo.SmallRows        SELECT
(                                BusinessEntityID,
Id,                              LastName,
LastName,                        FirstName,
FirstName,                       MiddleName
MiddleName                       FROM Person.Person;
)
```

Storage: example

Undocumented, unsupported feature shows page id for each line in result set:

```
SELECT sys.fn_PhysLocFormatter(%%physloc%%) AS [Row_Locator],
Id FROM dbo.SmallRows;
```

	Row_Locator	Id
19	(1:22517:18)	10314
20	(1:22517:19)	16699
21	(1:22517:20)	10299
22	(1:22517:21)	1770
23	(1:22517:22)	4194
24	(1:22517:23)	305
25	(1:22517:24)	16691
26	(1:22519:0)	4891
27	(1:22519:1)	10251
28	(1:22519:2)	16872
29	(1:22519:3)	10293
30	(1:22519:4)	4503

```
8*1024 bytes/page
-------------------- ~ 25 rows/page
4+3*(100) bytes/row

1:22517 = page id
0-24: row in page
```

Storage: example

```sql
CREATE TABLE dbo.LargeRows
(
Id int NOT NULL,
LastName nchar(600) NOT NULL,
FirstName nchar(600) NOT NULL,
MiddleName nchar(600) NULL
);

INSERT INTO dbo.LargeRows          SELECT
(                                  BusinessEntityID,
Id,                                LastName,
LastName,                          FirstName,
FirstName,                         MiddleName
MiddleName                         FROM Person.Person;
)
```

Storage: example

Undocumented, unsupported feature:

```
SELECT sys.fn_PhysLocFormatter(%%physloc%%) AS [Row_Locator],
Id FROM dbo.LargeRows;
```

8*1024 bytes/page
-------------------- ~ **2** rows/page
4+3*(1200) bytes/row

	Row_Locator	Id
1	(1:23823:0)	285
2	(1:23823:1)	293
3	(1:24116:0)	295
4	(1:24116:1)	2170
5	(1:24117:0)	38
6	(1:24117:1)	211
7	(1:24118:0)	2357
8	(1:24118:1)	297
9	(1:24119:0)	291
10	(1:24119:1)	299
11	(1:24120:0)	121
12	(1:24120:1)	16867
13	(1:24121:0)	16901
14	(1:24121:1)	16724
15	(1:24122:0)	10263
16	(1:24122:1)	10312
17	(1:25008:0)	10274
18	(1:25008:1)	10292

Storage: example

```
SET STATISTICS IO ON;

SELECT
Id,
LastName,
FirstName,
MiddleName
FROM dbo.SmallRows;
```
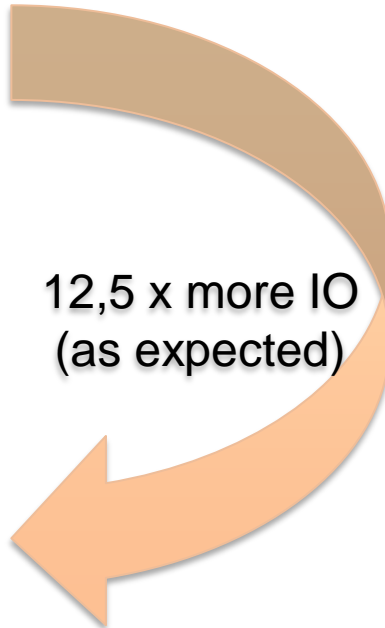
Messages:

(19972 row(s) affected)

Table 'SmallRows'. Scan count 1, logical reads 799, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

```
SET STATISTICS IO ON;

SELECT
Id,
LastName,
FirstName,
MiddleName
FROM dbo.LargeRows;
```

(19972 row(s) affected)

Table 'LargeRows'. Scan count 1, logical reads 9986, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

12,5 x more IO (as expected)

Storage: example

Query / Display estimated execution plan (also x 12.5)

(I/O cost has no unit, it a relative value)

Query 1: Query cost (relative to the batch): 100%
SELECT Id, LastName, FirstName, MiddleName FROM dbo.SmallR

Table Scan
Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Estimated Execution Mode	Row
Estimated Operator Cost	0.616362 (100%)
Estimated I/O Cost	0.594315
Estimated CPU Cost	0.0220477
Estimated Subtree Cost	0.616362
Estimated Number of Executions	1
Estimated Number of Rows	19972
Estimated Row Size	311 B
Ordered	False
Node ID	0

Query 1: Query cost (relative to the batch): 100%
SELECT Id, LastName, FirstName, MiddleName FROM dbo.LargeRo

Table Scan
Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Estimated Execution Mode	Row
Estimated Operator Cost	7.42155 (100%)
Estimated I/O Cost	7.39942
Estimated CPU Cost	0.0221262
Estimated Subtree Cost	7.42155
Estimated Number of Executions	1
Estimated Number of Rows	19972
Estimated Row Size	3611 B
Ordered	False
Node ID	0

Storage:conclusions

- Despite the fact that storage is cheap the data type has a huge impact on I/O.

- Keep this in mind at design time of your database tables; don't wait till queries get slow.

Partitioning

Use case

- In any time based application (accounting, payroll, time series, …) years of history are stored but >90% of all queries only involve last days or months.

- Also applicable to categorical data

- Partitioning offers a solution to fetch only necessary data.

Create table in filegroup

--Disk-Based CREATE TABLE Syntax

CREATE TABLE
   [database_name . [schema_name] . | schema_name .] table_name
   [AS FileTable]
   ({ <column_definition> | <computed_column_definition>
      | <column_set_definition> | [<table_constraint>]
| [<table_index>] [,...n] })
   [ON { partition_scheme_name (partition_column_name) | filegroup
      | "default" }]
   [{ TEXTIMAGE_ON { filegroup | "default" }]
   [FILESTREAM_ON { partition_scheme_name | filegroup
      | "default" }]
   [WITH (<table_option> [,...n])]
[;]

Create index in filegroup

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name
        ON <object> (column [ASC | DESC] [,...n])
        [INCLUDE (column_name [,...n])]
        [WHERE <filter_predicate>]
        [WITH (<relational_index_option> [,...n])]
        [ON { partition_scheme_name (column_name)
                | filegroup_name
                | default }]
                [FILESTREAM_ON { filestream_filegroup_name | partition_scheme_name | "NULL" }]
[;]

Partitioning

- Partition a table in different rowsets based on a column value, e.g. (but not restricted to) a date related field.

- Assign each partition to a separate file or filegroup.

- Older and rarely queried rows are separated from actual rows.

- Most queries need only actual rows → fewer rows have to be searched.

Partitions: example (db stackoverflow)

Use case: most users only want to see posts with a score of at least 4 or 5.

→   Use score as partitioning field

See script "partitioning.sql"

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