Ministerul Educației al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea de Calculatoare, Informatică și Microelectronică
Filiera Anglofonă ”Computer Science”

Definirea și gestiunea indecșilor

Lucrare de curs
Baze de date și cunoștințe

Efectuat de:
N.Maxian
Verificat de:
I.Cojanu

Chișinău 2017

Contents

List of Figures

List of Tables

Listings

Introduction

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

An index is a copy of selected columns of data from a table that can be searched very efficiently that also includes a low-level disk block address or direct link to the complete row of data it was copied from. Some databases extend the power of indexing by letting developers create indexes on functions or expressions.

Indexes are used in every database where performance matters. Although it has big benefit of performance improvement, we also have some consequences like:

– More memory is used

– Add/Update/Delete operations on indexed columns, invokes index rebuild and makes this operations to be slower.

1 Problem and domain analysis

1.1 Indexes

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

An index is a copy of selected columns of data from a table that can be searched very efficiently that also includes a low-level disk block address or direct link to the complete row of data it was copied from. Some databases extend the power of indexing by letting developers create indexes on functions or expressions.

1.2 Indexes Usage

1.2.1 Support for fast lookup

Most database software includes indexing technology that enables sub-linear time lookup to improve performance, as linear search is inefficient for large databases.

Suppose a database contains N data items and one must be retrieved based on the value of one of the fields. A simple implementation retrieves and examines each item according to the test. If there is only one matching item, this can stop when it finds that single item, but if there are multiple matches, it must test everything. This means that the number of operations in the worst case is O(N) or linear time. Since databases may contain many objects, and since lookup is a common operation, it is often desirable to improve performance.

An index is any data structure that improves the performance of lookup. There are many different data structures used for this purpose. There are complex design trade-offs involving lookup performance, index size, and index update performance. Many index designs exhibit logarithmic (O(log(N))) lookup performance and in some applications it is possible to achieve flat (O(1)) performance.

1.2.2 Policing the database constraints

Indexes are used to police database constraints, such as UNIQUE, EXCLUSION, PRIMARY KEY and FOREIGN KEY. An index may be declared as UNIQUE, which creates an implicit constraint on the underlying table. Database systems usually implicitly create an index on a set of columns declared PRIMARY KEY, and some are capable of using an already existing index to police this constraint. Many database systems require that both referencing and referenced sets of columns in a FOREIGN KEY constraint are indexed, thus improving performance of inserts, updates and deletes to the tables participating in the constraint.

Some database systems support an EXCLUSION constraint that ensures that, for a newly inserted or updated record, a certain predicate holds for no other record. This can be used to implement a UNIQUE constraint (with equality predicate) or more complex constraints, like ensuring that no overlapping time ranges or no intersecting geometry objects would be stored in the table. An index supporting fast searching for records satisfying the predicate is required to police such a constraint.

1.3 Index architecture/Indexing Methods

1.3.1 Clustered

Clustering alters the data block into a certain distinct order to match the index, resulting in the row data being stored in order. Therefore, only one clustered index can be created on a given database table. Clustered indices can greatly increase overall speed of retrieval, but usually only where the data is accessed sequentially in the same or reverse order of the clustered index, or when a range of items is selected.

Since the physical records are in this sort order on disk, the next row item in the sequence is immediately before or after the last one, and so fewer data block reads are required. The primary feature of a clustered index is therefore the ordering of the physical data rows in accordance with the index blocks that point to them. Some databases separate the data and index blocks into separate files, others put two completely different data blocks within the same physical file(s).

1.3.2 Non-clustered

The data is present in arbitrary order, but the logical ordering is specified by the index. The data rows may be spread throughout the table regardless of the value of the indexed column or expression. The non-clustered index tree contains the index keys in sorted order, with the leaf level of the index containing the pointer to the record (page and the row number in the data page in page-organized engines; row offset in file-organized engines).

In a non-clustered index,

The physical order of the rows is not the same as the index order. The indexed columns are typically non-primary key columns used in JOIN, WHERE, and ORDER BY clauses. There can be more than one non-clustered index on a database table.

1.4 MSSQL Execution Plan

A query plan, execution plan, or query execution plan is an algorithm showing a set of steps in a specific order that is executed to access data in a database

A query plan shows how a query was executed, or how it will be executed which is significant for troubleshooting query performance issues. Executing a SELECT statement to find out its query plan and effect on SQL Server performance can be acceptable, but executing UPDATEs to find that out is not an option. The plan is calculated by a SQL Server component Query Optimizer using minimum of server resources. When creating the SQL Server query execution plan, the number of database objects involved, joins, indexes and their availability, number of output columns, and more is considered

When a new query is executed, Query Optimizer evaluates the query plan, optimizes and compiles it, and stores it in the plan cache. The plan cache is a part of SQL Server buffer where data and query plans are stored (buffered), so they can be reused later

When a query is executed, Query Optimizer first searches the plan cache looking for a query plan that can be reused, thus making the execution faster. If there's no query plan that can be reused, a new one has to be created, which takes time and therefore makes query execution last longer

A very useful characteristic of query plans is that when a stored procedure is executed, the query plan is created for the stored procedure name and the same query plan will be reused whenever the stored procedure is executed, despite the values specified for procedure parameters. When it comes to executing ad hoc queries, query plans are created based on complete code, so different parameters or any change in code will prevent reuse of the existing plan. This clearly indicates what should be done to make your code run faster – wrap it up as stored procedures or functions, and the existing query plans will be reused and therefore code will be executed much faster

The slow execution of ad hoc queries can be mitigated by using the Optimize for ad hoc workloads option, introduced in SQL Server 2008. The option optimizes the plan cache use, as it solves the situation when query plans executed only once fill up the plan cache. As the buffer cache is used for both data and plan buffering, and the percentage of cache used for each changes in time depending on the current situation, it's necessary to use the cache wisely. Instead of buffering the whole plan, when the option is set to "True", only a fragment of the plan is buffered when the query is executed for the first time. When an ad hoc query is executed for the second time, its complete plan is buffered in the cache

1.5 MSSQL Execution Plan Elements

Since we are using Indexes for optimization of database, we need to know great tools given by Microsoft like Execution Plan. Execution Plan tells us what happens behind of query, what operations are done,what cost has every operations and how are merged together. We will use this functionality for analyzing effects of Indexation.
Although execution plan has a lot of elements, in this course work we will cover only few definitions such as Scan,Seek and types of Joins.

1.5.1 Scan

Since a scan touches every row in the table, whether or not it qualifies, the cost is proportional to the total number of rows in the table. Thus, a scan is an efficient strategy if the table is small or if most of the rows qualify for the predicate.

1.5.2 Seek

Since a seek only touches rows that qualify and pages that contain these qualifying rows, the cost is proportional to the number of qualifying rows and pages rather than to the total number of rows in the table.

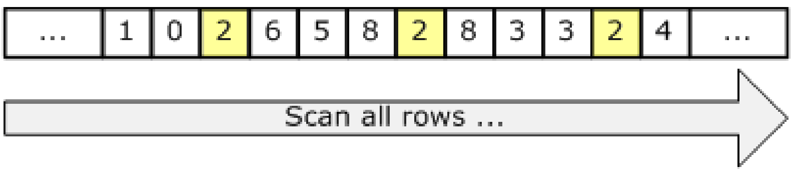


Figure 1.1 – Scan algorithm

Index Scan is nothing but scanning on the data pages from the first page to the last page. If there is an index on a table, and if the query is touching a larger amount of data, which means the query is retrieving more than 50 percent or 90 percent of the data, and then the optimizer would just scan all the data pages to retrieve the data rows. If there is no index, then you might see a Table Scan (Index Scan) in the execution plan.

Index seeks are generally preferred for the highly selective queries. What that means is that the query is just requesting a fewer number of rows or just retrieving the other 10 (some documents says 15 percent) of the rows of the table.

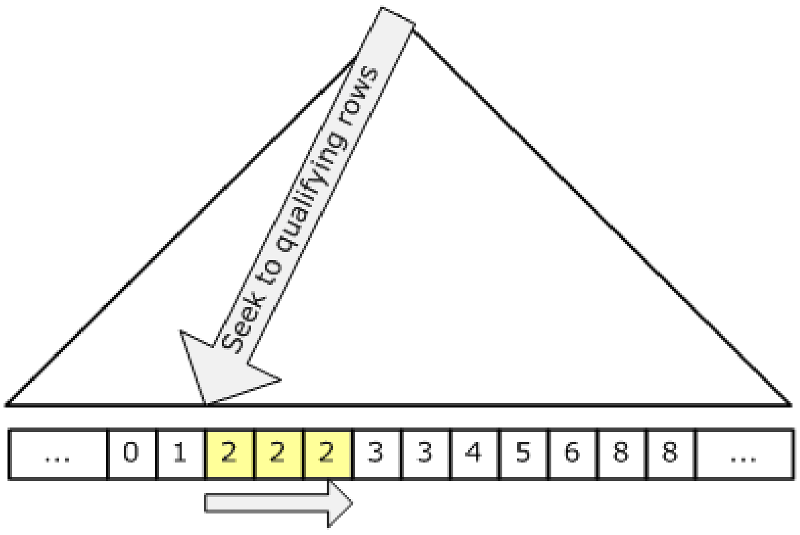


Figure 1.2 – Seek algorithm

1.5.3 SQL Joins [2]

SQL Server employs three types of join operations:

- Nested loops joins
- Merge joins
- Hash joins

If one join input is small (fewer than 10 rows) and the other join input is fairly large and indexed on its join columns, an index nested loops join is the fastest join operation because they require the least I/O and the fewest comparisons. For more information about nested loops, see Understanding Nested Loops Joins.

If the two join inputs are not small but are sorted on their join column (for example, if they were obtained by scanning sorted indexes), a merge join is the fastest join operation. If both join inputs are large and the two inputs are of similar sizes, a merge join with prior sorting and a hash join offer similar performance. However, hash join operations are often much faster if the two input sizes differ significantly from each other. For more information, see Understanding Merge Joins.

Hash joins can efficiently process large, unsorted, non indexed inputs.
For examples we can look into [4].

1.6 Problem definition

Main scope of this course work is to demonstrate indexation and it's benefits.So we will divide our problem into two parts.

- Clustered Index
- Non Clustered Index

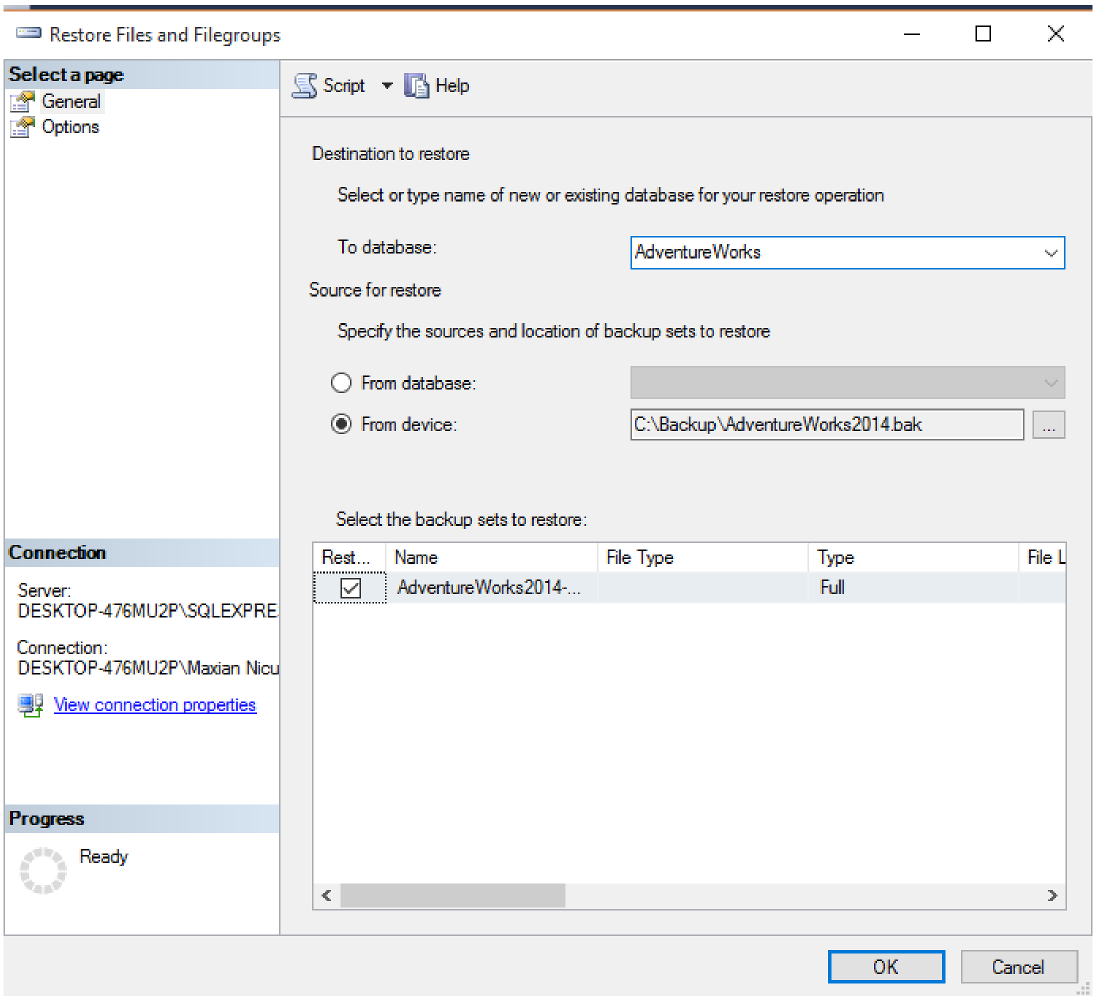
Both subproblems will have their own purpose and own examples. Each subproblem has an definition,analysis,solution and post-analysis.

2 MSSQL implementation. Particularities

2.1 Preparing database

2.1.1 Downloading database

In this Course Work I will use database Adventure Works [1].


Restore Files and Filegroups

Select a page: General, Options

Destination to restore

Select or type name of new or existing database for your restore operation

To database: AdventureWorks

Source for restore

Specify the sources and location of backup sets to restore

From database:

[✓ From device:] C:\Backup\AdventureWorks2014.bak

Select the backup sets to restore:

Rest...	Name	File Type	Type	File L
✓	AdventureWorks2014-...		Full	

Connection

Server:
DESKTOP-476MU2P\SQLEXPRE

Connection:
DESKTOP-476MU2P\Maxian Nicu

View connection properties

Progress

Ready

Figure 2.1 – Restoring AdventureWorks database

2.1.2 Droping database indexes

In order to show why we should use indexes, let remove all indexes for following tables:

- Production.Product
- Production.ProductCategory
- Person.EmailAddress
- Person.Password
- Person.Person
- Person.PersonPhone

– Person.PhoneNumberType

2.2 Clustered Indexes

2.2.1 Problem definition

Let's assume we have following query, which extracts ProductId, Name,Category and Standart-Cost from database.

```sql
SELECT p.ProductID ,

     p.name ,

       pc.name AS category ,

       p.standardcost

FROM   production.product p

       INNER JOIN production.productcategory pc

               ON p.productsubcategoryid = pc.productcategoryid

WHERE  p.productid = 760
```

Listing 2.1 – Finding product with a given id

2.2.2 Analysis of execution plan

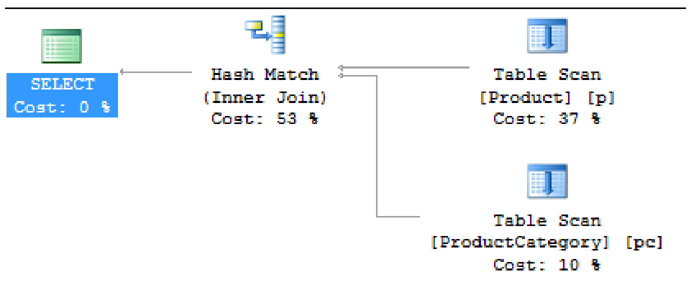


Figure 2.2 – Execution plan for the query

From this execution plan, we see that our query has cost of **0.034**. Furthermore, since our tables have no clustered indexes, data is stored as a heap, in the file. We see that our algorithm scans table for finding needed row. This means that query takes row by row and tries to find needed row that matches our criterions. This is ok for few rows, but when we have thousands of rows, performance will hurt us.

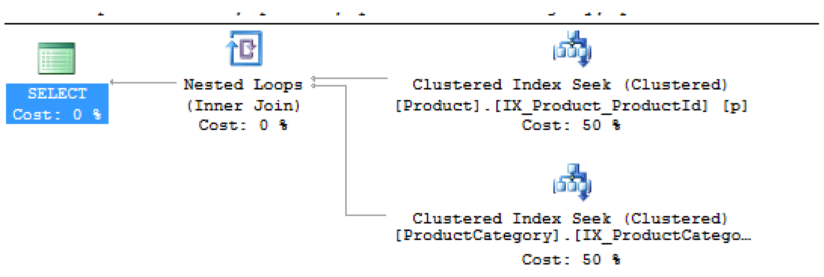
2.2.3 Solution

Of course Clustered Indexes. Clustered Indexes rearranges our table data as a b-tree structure. Let's create clustered index on Product table, with column ProductId. Now we can find needed product at once, without scanning all entire data ! Once we found product, we have also ProductSubCategoryId, which is also an clustered index for ProductCategory table, so we access that table also by id.

```sql
CREATE CLUSTERED INDEX IX_Product_ProductId
  ON Production.Product(ProductId)
CREATE CLUSTERED INDEX IX_ProductCategory_ProductCategoryID
  ON Production.ProductCategory(ProductCategoryID)
```

Listing 2.2 – Creating clustered indexes

2.2.4 Final results



SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Clustered Index Seek (Clustered)
[Product].[IX_Product_ProductId] [p]
Cost: 50 %
Clustered Index Seek (Clustered)
[ProductCategory].[IX_ProductCatego...
Cost: 50 %

Figure 2.3 – Execution plan after creating clustered indexes

Now we can see that we have **Clustered Index Seek**. This is faster way to find a data. This means we have no more loops, scan and everything that costs us data. Having ProductID, we can directly access Address of data, where row is situated. So, we have optimized our query as much as possible ! Now we have total cost of **0.0065**, 80% less than initial. That is awesome !

2.3 Non Clustered Indexes

2.3.1 Problem definition

Here we will try to extract basic data such as FirstName,LastName,Email,Password,PasswordHash,Ph
for each Person based on following tables:

- EmailAddress
- Password
- Person
- PersonPhone
- PhoneNumberType

```sql
SELECT p.BusinessEntityID,

       p.FirstName,

        p.LastName,

        ea.EmailAddress,

        pwd.PasswordHash,

        pwd.PasswordSalt,

        phone.PhoneNumber,

        phoneType.Name as PhoneType

FROM    person.person p

        INNER JOIN Person.Password pwd

                ON p.BusinessEntityID = pwd.BusinessEntityID

        INNER JOIN Person.emailaddress ea

                ON p.BusinessEntityID = ea.BusinessEntityID

        INNER JOIN Person.PersonPhone phone

                ON p.BusinessEntityID = phone.BusinessEntityID

        INNER JOIN Person.PhoneNumberType phoneType

                ON phone.PhoneNumberTypeID = phoneType.PhoneNumberTypeID
```

Listing 2.3 – Selecting person base information

2.3.2 Analysis of execution plan

Hence, we obtained an cost of **5.94** for our query with 19972 rows.

Since we don't have any index here, we can observe that in execution plan we have only "Table scan". This means in our query, SQL engine scan table row by row, which is very slow.

2.3.3 Solution

In order to optimize it, let's to improve our queries with Non Clustered indexes. This means that, instead of big tables we will search into smaller index tables, which contains less data and can be found with a given index.

Since we have 13 columns on **Person** table, it make us to have a big overhead when extracting data. Since we assume we will execute a lot of times this query, we need to optimize this. Non-clustered indexes are great for this. We could "shrink" our data table by creating smaller index table. We will create index on **BussinessEntittyID** column because we use this column in JOIN clauses. We also need **FirstName** and **LastName** information from that table, we also define this columns

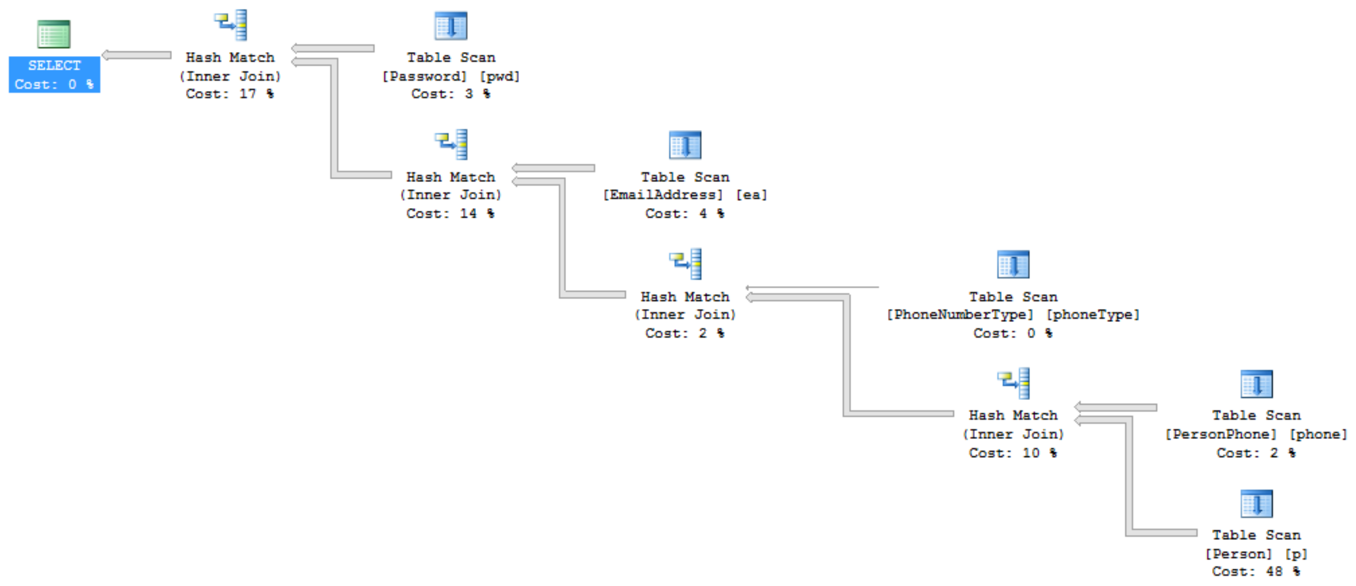


Figure 2.4 – Execution plan for the query

to be included into non-clustered index table. Same things we will do for each table, in order to optimize our query and remove overhead which we acutally don't need.

```sql
CREATE NONCLUSTERED INDEX IX_Person_BusinessEntityID_FirstName_LastName
    ON Person.Person (BusinessEntityID) INCLUDE(FirstName,LastName)

CREATE NONCLUSTERED INDEX IX_Password_BusinessEntityID_PasswordHash_PasswordSalt
    ON Person.Password (BusinessEntityID) INCLUDE(PasswordHash,PasswordSalt)

CREATE NONCLUSTERED INDEX IX_EmailAddress_BusinessEntityID
    ON Person.EmailAddress (BusinessEntityID) INCLUDE(EmailAddress)

CREATE NONCLUSTERED INDEX
   IX_PersonPhone_BusinessEntityID_PhoneNumberTypeId_PhoneNumber
    ON Person.PersonPhone (BusinessEntityID,PhoneNumberTypeId) INCLUDE(
   PhoneNumber)

CREATE NONCLUSTERED INDEX IX_PhoneNumberType_BusinessEntityID_Name
    ON Person.PhoneNumberType (PhoneNumberTypeId) INCLUDE(Name)
```

Listing 2.4 – Creating non-clustered indexes

2.3.4 Final results

Now, we see that instead of table scan we have Index Scan. Which is faster than what we had. Also now we have in indexes everything what we need, so no need to lookup into the table for missing columns.

Now we have an cost of **3.12**, which is 48% less than we had before. Nice ! It's an improvement.

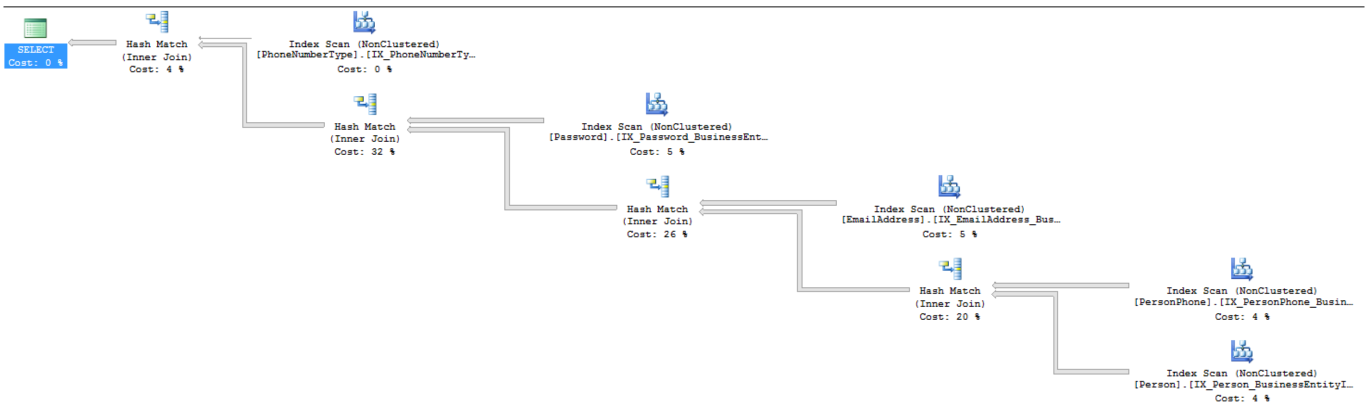

SELECT
Cost: 0 %
Hash Match
(Inner Join)
Cost: 4 %
Index Scan (NonClustered)
[PhoneNumberType].[IX_PhoneNumberTy…
Cost: 0 %
Hash Match
(Inner Join)
Cost: 32 %
Index Scan (NonClustered)
[Password].[IX_Password_BusinessEnt…
Cost: 5 %
Hash Match
(Inner Join)
Cost: 26 %
Index Scan (NonClustered)
[EmailAddress].[IX_EmailAddress_Bus…
Cost: 5 %
Hash Match
(Inner Join)
Cost: 20 %
Index Scan (NonClustered)
[PersonPhone].[IX_PersonPhone_Busin…
Cost: 4 %
Index Scan (NonClustered)
[Person].[IX_Person_BusinessEntityI…
Cost: 4 %

Figure 2.5 – Execution plan after non-clustered index creation

3 Practical part

3.1 Differences and Similarities between MSSQL and Oracle

Since MSSQL was base SQL Database for my course work, here I will define things that are different in SQL Database.

3.1.1 Execution plan

In SQL Developer to show execution plan, you should press F10 or click the Explain Plan icon. Of course it looks different and has it's features which makes him different of what we have in SSMS.

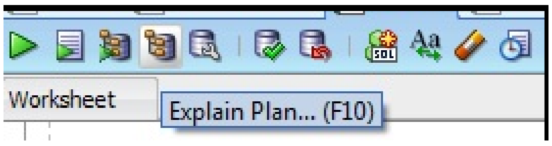


Figure 3.1 – Showing execution plan

3.1.2 Changes index creation queries

Because Oracle SQL doesn't have **INCLUDE** statement on creating indexes. To migrate our queries we need only to change queries which have **INCLUDE** statement.

```sql
/* MSSQL Syntax */
CREATE NONCLUSTERED INDEX IX_Password_BusinessEntityID_PasswordHash_PasswordSalt
    ON Person.Password (BusinessEntityID) INCLUDE(PasswordHash,PasswordSalt)

/* Oracle Syntax */
CREATE NONCLUSTERED INDEX IX_Password_BusinessEntityID_PasswordHash_PasswordSalt
    ON Person.Password (BusinessEntityID,PasswordHash,PasswordSalt)
```

Listing 3.1 – Finding product with a given id

Conclusions

From my point of view, Indexation should be know by everyone who is planning and developing a database. Indexes gives us a better performance for our database and queries which are important for you, and needs to be optimized. By defining indexes, we are creating new index table which has references to original data table, or could include data directly in it, as we have done in our case. Of course, we cannot make everything an index ! After changing table data or changed columns, SQL will perform an index rebuild ! We need to know that rebuild = time. So, we need to make sure that we have enough indexes to make database faster and not many to make it work slower due the indexes rebuild. Indexes are key factor for performance, once you have a good database architecture.

Also I would say that MSSQL Execution Plan is a very very useful feature for SSMS. It shows us every steps what SQL is doing in order to give you an result. Execution Plan gives you imagination on what happens behind the query. Using this tool we can optimize our queries and indexes.

References

1 Adventure Works 2014 - Full database `https://msftdbprodsamples.codeplex.com/downloads/get/880661`

2 Advanced Query Tuning Concepts `http://msdn.microsoft.com/en-us/library/ms191426\%28v=SQL.100\%29.aspx`

3 Index Seek vs Index Scan `https://blog.sqlauthority.com/2007/03/30/sql-server-index-seek-vs-index-scan-table-scan/`

4 LOOP, HASH and MERGE Join Types `http://www.madeiradata.com/loop-hash-and-merge-join-types/`