Identify and implement index changes for queries

We're going to have a look at identifying and implementing index changes for queries, together with assessing index design for performance.

What is index?

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

Note: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

The Index in SQL is a special table used to speed up the searching of the data in the database tables. It also retrieves a vast amount of data from the tables frequently. The INDEX requires its own space on the hard disk.

The index concept in SQL is same as the index concept in the novel or a book.

It is the best SQL technique for improving the performance of queries. The drawback of using indexes is that they slow down the execution time of UPDATE and INSERT statements. But they have one advantage also as they speed up the execution time of SELECT and WHERE statements.

In SQL, an Index is created on the fields of the tables. We can easily build one or more indexes on a table. The creation and deletion of the Index do not affect the data of the database.

Why SQL Index?

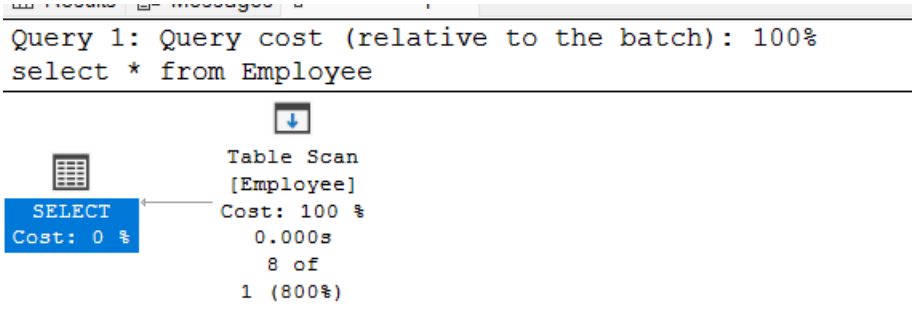
The following reasons tell why Index is necessary in SQL:

- **SQL Indexes can search the information of the large database quickly.**
- **This concept is a quick process for those columns, including different values.**

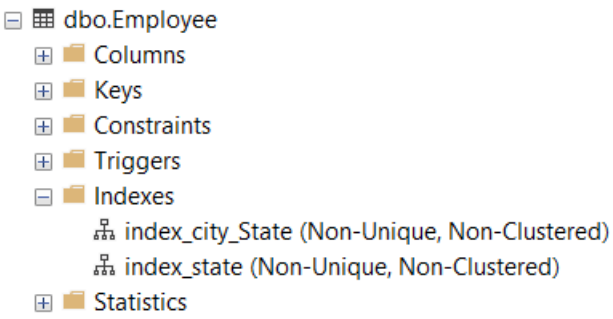
- **This data structure sorts the data values of columns (fields) either in ascending or descending order. And then, it assigns the entry for each value.**
- **Each Index table contains only two columns. The first column is row_id, and the other is indexed column.**
- **When indexes are used with smaller tables, the performance of the index may not be recognized.**

Example for creating an Index in SQL:

```
-- Table Creation
create table Employee (
Emp_ID int,
Emp_Name varchar(20),
Emp_Salary int,
Emp_City varchar(25),
Emp_State varchar(25)
)

-- Adding content into table
insert into Employee values (101,'Veera',12000,'Chennai','Tamil Nadu')
insert into Employee values (102,'pranesh',12000,'Chennai','Tamil Nadu')
insert into Employee values (103,'kavi',12000,'Chennai','Tamil Nadu')
insert into Employee values (104,'praveen',12000,'Chennai','Tamil Nadu')
insert into Employee values (105,'pravin',12000,'Chennai','Tamil Nadu')
insert into Employee values (106,'ananya',12000,'Chennai','Tamil Nadu')
insert into Employee values (107,'RamaPraba',12000,'Chennai','Tamil Nadu')
insert into Employee values (108,'Priya',12000,'Chennai','Tamil Nadu')

-- Reading from table
select * from Employee
```



```sql
-- Index creation

CREATE INDEX index_state ON Employee (Emp_State);  -- one column index
CREATE INDEX index_city_State ON Employee (Emp_City, Emp_State);  -- two column index
```



So, they stop us from having to go through the entirety of a table and allow us to seek a particular point.

So, what are the requirements for indexes?

Well, first, a big table.

If you've got a small table, you probably don't need an index. Small tables, even if you provide an index, may just use the scan anyway.

You should have a small column size,

So, the best are things like numeric, but, if you've got others like smaller text sizes, then that could be good as well.

So having an index on an Nvarchar 60, probably not that good.

You should use columns which are in the where clauses and they need to be sargable.

So, in other words, we've had a look at why you're using functions like year and left and other things, they just don't work with indexes with sargs.

Whereas if you used other things, those are sargables,

so less than, greater than, that sort of thing.

So, if you're using a like, then this is perfectly well-formed sargable where clause.

Search ARGument ABLE

www.linkedin.com/in/ragasudha-m

A sargable query is a query that uses search arguments to efficiently narrow down results using indexes. The term is short for "Search ARGument ABLE".

How does it work?

- **The database engine can use an index seek to speed up the query execution.**
- **The query uses search arguments that allow the database engine to find and efficiently narrow down the results using indexes.**

Why is it important?

- **Sargable queries are important because they can lead to a good query plan.**
- **They can help avoid complex, time-consuming cost-based searches.**
- **They can help improve query performance.**

How to make queries sargable?

- **Avoid using functions in the WHERE clause that operates on column values.**
- **Use search arguments that allow the database engine to use indexes.**

What happens when a query is not sargable?

- **A query that is not sargable is known as a non-sargable query.**
- **Non-sargable queries can negatively affect query time.**
- **Query optimization can help convert non-sargable queries to sargable queries**

So, where HI's at the beginning.

However, if I said where HI is anywhere, well, I can't use an index for that, because I would have to go through all the items of the index anyway.

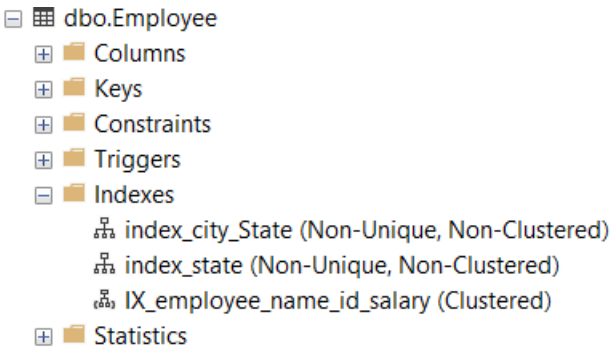
So how do you create an index?

Well, in TSQL, you would say 'create'. And then, you would either have a non-clustered, or a clustered index.

www.linkedin.com/in/ragasudha-m

So, name of index, I usually put IX_ and then the table and then an _ and then the individual columns that it was indexing, because the clustered or non-clustered index name needs to be unique.

```
create clustered index IX_employee_name on dbo.employee
(
emp_name
)
```



So, create, clustered or non-clustered, index, the name of the index, on, and then the name of the table, and then in brackets, that's where you put the columns.

Now what is a clustered index as opposed to a non-clustered index?

Well, you're only allowed one clustered index per table. It's frequently used with primary keys, when you create a primary key, you also create, automatically, a clustered index.

It re-sorts the table based on the index. So, a heap is a table without a clustered index,

Once you have a clustered index, it gets re-sorted.

So, you should use it for frequently used queries and range queries.

So, between x and y. Clustered indexes, when created with primary keys,

use the unique clustered index. So that means that there can only be one row with each value. If you're using multiple columns, then it's only one row, but a combination of the values. It is possible to create a non-unique clustered index, most of the time it will be unique.

So, these are things that should be accessed sequentially, in ranges, it's quite good for identity columns. So, identity columns are automatically created data columns.

So, it starts off 1, 2, and 3, in other words, it's a sequential numbering. You don't specify it, the computer does, and clustered indexes are frequently used.

So, whereas clustered indexes re-sort the table, and you can only have one per table because you can't sort a table two different ways at the same time, you can have as many non-clustered indexes as you want.

It creates a separate index. But be warned, you insert row, update a row, delete a row, merge data sets together then all of the indexes will need to be adjusted.

So, if you've got too many indexes, that could be slowing down your machine. Now you don't have to index the entire table.

Suppose you have a where, a frequently used query, with a very specific where. So, where the city is equal to Bothell. Well, you could have an index that looks at just that where clause.

That's called a filtered index.

```
create nonclustered index ix_address_city_where on saleslt.address
(
addressline1,
addressline2
)
where city ='bothell'
```

Now let's imagine a hypothetical index, so I'll create this index, and it's got references to all these rows, and this happens on a particular page. And then there's a new page which has everything else. And then another page which has everything else as well.

So, each of these are on separate but linked pages, and there's also a hierarchy which says go to this page if you want 1-8, this for 9-17, this for 18-25. And this is all that we can contain on a particular page in this example.

Now what happens if I insert row number 16? Well, I need to put it in here because an index needs to be in the right order. But I haven't got any room. So,

what I need to do is break the page and create a separate page, maybe 16 goes on this page, 17 goes on the next page. It doesn't then redo everything; it just creates a new page.

Now suppose you didn't want your index to use up all the available space because you knew that you were going to be adding additional information.

So, at the time of creating the index, you wanted it to say, just allow 5 for each page. So, it could have 8, in terms of capacity, but you're just saying, all I want is 5. Well, you can do that with something called the fill factor.

So, I can say with fill factor equals, and in this case it would be 5 divided into 8. This is about 62 per cent. So now if I have this index and row 15 comes along, then good news, here's my page, it's only got 5 items on it, I can just insert it, no need for the page to split, which obviously will take some time.

```
create nonclustered index ix_address_city_where on saleslt.address
(
addressline1,
addressline2
)
where city ='bothell'
with (fillfactor = 62)

-- 5/8*100% = 62.5%
```

And finally, we can say that it is going to be sorted ascending or descending, just like with an order by, so the default is ascending, if you happen to be using a particular query, lots of queries where it's descending, you might want to create the index that way.

So, this is how we can create an index in TSQL.

Now you can't do this for the Azure SQL database, but in other types of databases, you could in SSMS, go to the indexes part, right click, and go new index, and have a visual way of doing that.

As you can see it just gives you an index template for the Azure SQL database. So, if I connect to a different database, I'm going to connect to my local database, so this could be like an SQL server on an Azure virtual machine, and go to a particular table, and right click on indexes and go new index. Here you can see we can create clustered indexes and non-clustered indexes and so forth and have a dialogue box to do that.

But that is not available in Azure SQL database. Just a quick note about column store. Now we won't be talking too much about column store, but just to let you know, the traditional way of having data stored, so like this, is called row store. So, we have each row contained within a set unit.

In SQL server 2012, they introduced column store, and it wasn't that good in terms of number of situations when you can use it in 2012, but it got a lot better in 2014 and then got even better in 2016. So, what column store does, is it store each column separately, and then you must get it together at the end.

So, for instance, just like you get ranges of rows together, well it's the same way being able to get columns together. So, the advantage of this, if you've got a huge amount of data, your data warehouse we're talking about, wouldn't it be a lot easier to be able to just say okay, give me all the rows with this city?

And what a column store table does, and therefore a column store index does, is it concentrates on a particular city, or anything else in a page, and it compresses it down, so it says this is city 1, 2, 3, 4, 5, 6 and it has a list of all of the meanings of what city 1 and 2 and so forth are in the page header.

We'll be looking at column store later when we talk about the use of compression for tables indexes. But if you see what column store is, it's not the standard type of index. It's an index on a different type of table.

However, column store indexes are generally available in most tiers of Azure database. So, creating a table. Use creates, clustered or non-clustered, or unique clustered, you could have unique non-clustered if you want, index, name of the index, on, name of table, and then in brackets the columns.

You could have a filtered index if you used a where clause and you can have spaces in the index, so it takes more pages, but there's less on each page, if you use the fill factor. And that is expressed as a percentage except you don't use a percentage sign. So, it goes from 1 to 100.

And finally, if you no longer need an index you can always drop it. If I right click on an index and go to script index as drop, you can see it's a very simple.