**A quick search without an index**

[Alex Chamchourine](https://www.sqlservercentral.com/author/Alex-Chamchourine), 2020-02-14

FROM: <https://www.sqlservercentral.com/articles/a-quick-search-without-an-index?utm_source=ssc&utm_medium=pubemail>

The Issue at hand

We all have days at work when someone comes to us with that truly impossible request which will require a miracle to accomplish. Mine happened when a coworker from Marketing approached me with a seemingly simple one: if I could fetch some data from one table from a specific month a couple of years back. Not a big deal, one would say, but I vaguely remembered that the table was on the large side. It had a creation datetime column but did it have an index on it?

Of course, they also wanted the data quickly. I said the usual, “I will see what I can do,” and went to have a closer look at the table in question. The luck never misses us, the index indeed was non-existent and the table was huge. Not a problem, we could scan the table, right? Wrong. If I’ve learned anything over years of working with databases, it’s that size does matter. A table that has hundreds of millions of records, consisting of a few integer columns, would be formidable enough. Then, throw in various varchar and datetime columns. Now that’s a real challenge, isn’t it?

The table I was looking at had a billion records, literally. To merely scan the table could have easily taken more than a day, and I needed to process the pulled data as well. Also, scanning a table that size might end up not being as benign to the overall system health as it seems at first glance. All the scanned pages have to be pulled from the disks to the Sql Server memory, flooding it. The flood, in turn, would flush data pages that could be used for current operational tasks. Your current user’s requests might have to wait longer while the server is re-reading data from the disks, instead of quickly reusing data pages from the memory. The performance might suffer, and in the worst case scenario the server can be brought completely to its knees . So, doing a table scan, one should exercise a special technique – running it in small batches, while storing the current position and intermediate results somewhere. This approach allows the server to readjust and have some breathing room.

And there I was, contemplating the script for the job and dreading the time it would take to run, when it occurred to me that creation datetime values were aligned with the table ID. The ID was an identity column, with ever-increasing values and the primary key built upon it. Being aligned with the ID, the creation datetime values were naturally pre-sorted in the same way as the ID values were. Hold on, I thought, I can implement something enormously faster than the table scan, something, that in the worst case scenario, would have required only 30 steps instead of 500,000,000! The binary search!

The Search

For all of you who, like me, graduated quite some time ago, a refresher on the theory is in order. The binary search is a simple, yet extremely efficient, way to search for a value in a sorted array (a table column in our case). The array must be pre-sorted and finite. The search is done by comparing the middle element of your array with the target. If they are equal – the search is over. If they are not, you eliminate the half in which the target cannot be and repeat the previous step to the remaining one. Find the middle, compare the target with it, if they are not equal, again eliminate a half and so on. The total number of steps required to find the target in the worst case scenario, when you find it by the very last step, will be logN, where N is the number of elements. Compare it to N/2 when you are just iterating through. Generally speaking, it would be N but, if we can guess whether the target is closer to the end, we could go backwards, thus reducing the total number of required steps.

In my case, N=1,000,000,000 and that’s how I came to the two numbers above: 30 and 500,000,000. The ID would have played the role of an array enumerator and the creation datetime would have been the array element value. There was one caveat though. The array enumerator, by its very definition, is a continuous gapless sequence of integer numbers. The table IDs might have easily had some gaps, due to record deleting or ID reseeding. Simply defining the middle by dividing the ID range by 2, one should not expect that the record with that ID will be there. Instead of a direct search, I had to use the top() function. Something like this:

Select top(1) \* from Table where id <= @id order by id desc

I used “<=” and “desc” because I wanted to find the value either equal to or immediately before the target. Given that @l, @r were the left and right boundaries respectively, @id was the middle, @dt was the creation datetime, @tdt was the target value and @idr the real table ID, the algorithm could look like the following:

while @l <@r

begin

--find the middle

@id = @l +floor((@r-@l)/2)

--find the table record

select top(1) @idr = id, @dt = creation\_datetime from Table where id <= @id order by id desc

--move the boundary

if(@dt<@tdt)

@l = @id +1

else

@r = @id

end

The last found @idr would have been the ID of the record I was after. The algorithm had something of a “left” bias, that is, a tendency to choose the leftmost of all values. Since I wanted the record with a value as close to the target as possible, I also checked the nearest left and right neighbors in the table to see whether one of them could have been a better match. Note that I did not use the real ID from the table for the iteration process, as in that case, with gaps in ID values and under some circumstances, the algorithm could have ended up cycling between two states indefinitely.

The writing and testing of the script took me about an hour. By using it I found the first record with the particular creation datetime. After that I used a simple select statement with a where clause that contained both conditions: *id >= @<found id> and creation\_datetime >= @dt1 and creation\_datetime < @dt2*. I only needed to make sure the optimizer would have chosen to use the primary key instead of the index or table scan. All in all, I was done in less than 2 hours! It was fascinating to find once more that computer science is not something esoteric, buried deep inside sql server assemblies, but rather something that can be easily employed in day to day work.

Below is the whole script I wrote. Please see the comments inside to understand how to use it.

/\*

Run this script in your database

It will find the value either equal to or immediately before the target.

Note that datetime precision is limited to 3 ms

\*/

-- debug flag, if set, it will show the results for every iteration

declare @debug bit = 0

-- @Table is the name of the table you want to search in.

declare @Table varchar(128) = 'TableX'

-- @Id is the name of identity (id) column for the table

declare @ID varchar(128) = 'Id'

-- @DateTimeColumn is the name of your creation datetime column in the table

declare @DateTimeColumn varchar(128) = 'created\_dt'

-- this is the target datetime value

declare @TargetDateTime datetime = '2020-01-03 18:23:03'

declare @Sql varchar(max)

set @sql = '

-- this is your debug flag

declare @debug bit = <debug>

-- this is your target value

declare @tdt datetime = ''<TargetDateTime>''

-- this is the variable we store the middle value (the result of dividing)

declare @id bigint

--this is your left and right boundaries respectively

declare @l bigint, @r bigint

-- these are the variables we store the results of the current step search, datetime and table id respectively

declare @dt datetime, @idr bigint

-- find first "left" and "right" values

select @l = min(<ID>), @r = max(<ID>) from <Table>

while @l < @r

begin

-- expected id

set @id = @l +floor((@r-@l)/2)

-- find the value and id of the record

select top(1) @dt = <DateTimeColumn>, @idr = <ID> from <Table> where <ID> <= @id order by <ID> desc

-- if debug is requested show current values

if( @debug = 1 )

begin

select ''@id'' = @id, ''@l'' = @l, ''@r'' = @r, ''@dt'' = @dt, ''@idr'' = @idr

end

if(@dt < @tdt)

set @l = @id +1

else

set @r = @id

end

-- check if the neighbouring records have a better match

declare @t table(id bigint, dt datetime, dlt float)

insert @t(id, dt, dlt)

select top(1) <ID>, <DateTimeColumn>, abs(cast(<DateTimeColumn> as float) -cast(@tdt as float))

from <Table>

where <ID> < @idr

order by <ID> desc

insert @t(id, dt, dlt)

select top(1) <ID>, <DateTimeColumn>, abs(cast(<DateTimeColumn> as float) -cast(@tdt as float))

from <Table>

where <ID> > @idr

order by <ID>

insert @t(id, dt, dlt)

select @idr, @dt, abs(cast(@dt as float) -cast(@tdt as float))

select top(1) @dt = dt, @idr = id from @t order by dlt, id

select ''Found Value'' = @dt, ''Record Id'' = @idr

'

set @sql = replace(

replace(

replace(

replace(

replace(@sql, '<ID>', @ID),

'<Table>', @Table),

'<TargetDateTime>', convert(varchar(50), @TargetDateTime, 121)),

'<debug>', @debug),

'<DateTimeColumn>', @DateTimeColumn)

exec (@sql)