Article
Kyle Baxter · Jan 11, 2019 10m read

The One Query Performance Trick You NEED to Know? Tune Table!

A good writer is supposed to draw you in with the title and bury the answer somewhere in the article. I suppose this makes me a bad writer – don ’ t think less of me, my whole feeling of self-worth comes from the opinions of internet strangers! When my dear colleague Brendan was reviewing information on the Developer Community he noticed that we failed to write anything on Tune Table! This tool is the 2nd most important piece of query performance (the first being indexes – no indexes, no speed), and it is worth taking the time to understand. I ’ m going to be using a lot of half-examples here, but the details can be filled in trivially. In all examples we can assume all fields are individually indexed unless stated otherwise.

Alright, suppose we have the following query:

```
SELECT *
FROM  People
WHERE HomeState  = 'MA'
    AND PersonId = '123-45-6789'
```

Which index should we use? Seriously, stop reading and come up with an answer. Say it out loud. Don ’ t worry, your colleagues probably won ’ t think you ’ re crazy.

The correct index to use is, of course, the index on PersonId. Why is this true? Well we know that the PersonId is a near-unique numeric(-ish) identifier while potentially millions of people reside in each state. What we have to understand is that we know this because we know some information about the schema based on the column identifiers. To a computer these names are meaningless. To illustrate this point, let ’ s look at this query:

```
SELECT *
FROM  TheTable
WHERE Field1 = 32
    AND Field2 = 0
```

Do we use the index on f1 or f2? The query above is similar to what the query optimizer has to work with. How do we pick indexes in this case? We need to know about the data in order to make good decisions. This is where TuneTable comes in.

Tune Table looks through a sampling of the data in your table and stores statistics about your table so that our query optimizer can make good decisions. If you have never run Tune Table, then your queries are effectively choosing indexes at random (this is not strictly true, but it might as well be). In fact, if you ever call support because of a slow query, the first thing we ’ re going to ask you is “ Have you run Tune Table? ” The answer better be ‘ yes ’ .

Running Tune Table

To run Tune Table you can go to a terminal and execute:

```
d $SYSTEM.SQL.TuneTable(<Table>)
```

There are many flags you can pass in, but I recommend:

```
d $SYSTEM.SQL.TuneTable(<Table>,1,1,.msg,1)
```

These flags update the class definition, display the new values, and the last one keeps the class up-to-date. You ’ ll need to compile all embedded queries against your tables and purge cached queries so that your queries can take advantage of the new Tune Table information. You can read more about the flags here:

https://docs.intersystems.com/latest/csp/documatic/%25CSP.Documatic.cls?...

What Tune Table Does

Tune Table has a couple of steps. First, it computes the number of rows in the table by running SELECT COUNT(*). With that number, it figures out how many rows it wants to use as a sample (it ’ s asymptotically similar to the square-root of the number of rows). Then, for each ID defined for the table, we flip an appropriately weighted electronic coin and determine if that row will be in the sample. We use this sample to calculate the statistics we can then use to optimize your queries.

There are two important bits here. The first is that we need to get the count. The second is that we need to be able to get each ID in your table. The result is that having an extent index drastically helps the run time of Tune Table. If your table is bitmap-friendly (that is, has a positive integer IDKEY) then any bitmap index will automatically give you a bitmap extent.

What does Tune Table measure? I ’ m going to break it up chronologically.

The Two OG ’ s

The following two metrics are the oldest Tune Table metrics. How old are they? They were added before I got to High School (1999). If you are running pre-Y2K Caché then please shoot me an email, because I would love to talk to you.

Extent Size – measures the number of rows in your table.

This metric is the easiest one to understand and is useful in deciding on JOIN order. Suppose we have the following query:

```
SELECT *
FROM   Table1 T1
    JOIN Table2 T2
     ON T1.Field1 = T2.Field1
```

Should we read T1 and then JOIN in the fields from T2 or vice-versa? ExtentSize will tell us which table has fewer rows and therefore which table we ought to start with. In the extreme case, you can imagine that T1 has 10 Billion rows while T2 has 100 (think something like T1 is a table of all visits to your hospital over all time, and T2 is your table of patients currently at your hospital). In that case you want to start in the small table and JOIN in the larger one, as JOIN restricts the number of rows you have to read in.

Selectivity – measures the uniqueness of a value for a given field.

This metric takes a little more explaining. There are 2 ways to think of it:

1. 1/x where x is the number of possible values the field can take
2. Average percentage of the table returned for the query SELECT * FROM T WHERE field=?

Let ’ s look at some examples to help clear this up, because neither of these explanations make sense.

For the above query we had a HomeState field. That would (in the USA) necessarily have a selectivity = 2.0%. Why? Well there are 50 states so 1/50 = .02 = 2%. That ’ s easy enough. But what about distribution? Clearly people are not evenly distributed over states! This is true, but selectivity doesn ’ t care about that. Uniform distribution ends up being a good estimation for most queries and it ’ s easy to calculate meaning queries compile faster.

Another example, let ’ s think about PersonId . a PersonId might, occasionally, have to get reused as people die and are born, so the number isn ’ t quite unique. However, it is very very unique. For that we might see selectivity = .00001%. That is a very good selectivity! The query SELECT * FROM People WHERE PersonId = ? will bring back one or two rows each time. In general, the lower the selectivity the better. There is an exception…

What if you have a unique field – the ID is a good example. The selectivity of a unique field is 1. Note, every selectivity I ’ ve mentioned so far is a percentage. 1 is not a percentage but a number of rows. So the query: SELECT * FROM People WHERE ID = ? will always return 1 row.

That ’ s a lot of information about one field, but it ’ s perhaps the most important one. The rule is: the lower the percentage the better, and 1 is the best value of all.

The One We “ Always ” Had

The following statistic was always estimated but started being explicitly measured in 2013.1

Block Count – a measure of how many blocks each map (index & the master map) take up on disk.

We have been estimating block count for a long time, but we decided to measure it explicitly back in 2013. While this seems like it might be redundant along with Extent Size, it has other utility. Here ’ s a good query to explain:

SELECT COUNT(*) FROM MyTable

If every field is indexed then which one should we read? Well we want to read the one that takes up the least amount of space on disk. This metric gives us that information explicitly. It also pops up when trying to estimate the cost of reading chunks of indexes in more complicated situations. In general, it ’ s good at measuring the width of your maps, while ExtentSize measures the length.

Block count is also useful if you use a parent/child relationship. This will (by default) store both tables in the same global, and without Block Count the Query Optimizer wouldn ’ t have any idea that a map read on a potentially small parent table could actually be very large due to the large number of children. In fact, this is one reason we do not recommend using Parent/Child relationships in new development (EVER! If you think you want to, please call into Support and let us discuss it with you!).

Brendan ’ s Note/Trivia: We actually had ExtentSize, Selectivity, and BlockCount all the way back *to FDBMS days, so at least 1992, but you had to set them manually.*

The One That Changed Everything

In 2014.1 we added a metric to determine if one value of a field was over represented. This caused a big difference in how we calculated selectivity:

Outlier Selectivity - The selectivity of a field that is overrepresented in the table.

To understand a situation with an outlier, let ' s think about a hospital. Hospitals generally service their local community. For the field HomeState most of the values will be the same as the state the hospital is in. Let ' s consider a hospital in MA (because that ' s where I live). 90% of my local hospital ' s patients are from MA, while others have traveled from other states, or are visiting. We want to indicate that searching for HomeState= ' MA ' is not selective, while searching for any other HomeState IS selective. That ' s what outlier selectivity gets us.

As an example – Tune Table will calculate the Selectivity for the HomeState field outlined above as something like 0.04%, and indicate that the outlier selectivity is 90% and the outlier value would be ' MA ' . The Selectivity generally goes down. This change was a big one, as it changed the way selectivities were calculated and came up with some query plans that surprised some people. If you are upgrading across the 2014.1 boundary, keep this in mind.

The Last One

We added this last metric to better be able to estimate temp file sizes in 2015.2

Average Field Size – The average size of a field

This metric is easy to understand as well. While we are taking our samples it also computes the average size for each field in the table. This is useful for determining the size of temp files, and allows the optimizer to figure out if we can build temporary files in memory (local array) or if we need a disk-backed structure (process private global).

Practical Considerations

The next question you should have is “ how frequently do I need to run this? ” The answer is a bit subtle. You must (MUST MUST MUST!!!!) run it at least once, or provide some values yourself. After that, you might never have to run it again. The typical answer for this is that if you are happy with your performance now, there ’ s no reason to run it. Indeed, if you have a mature database, and your data is growing at a steady rate among your tables then you probably have good values going forward.

If, however, your tables are changing and your SQL performance is degrading, you might want to rerun Tune Table. If you are having a lot of data incoming and your tables are changing by orders of magnitude it might be time to run Tune Table. But this is all predicated by the situation of “ All my queries are slow ” . If you get to this point, please contact the WRC who will want to look at some of your system settings.

If you are going to run Tune Table, you want to make sure to run it on all tables that are related to the table you want to tune. Otherwise, their Extent Sizes and Block Counts become incomparable across tables (and this WILL make your query performance suffer).

If you are on a version newer than 2016.2 you have the advantage of Frozen Query Plans, which will allow you to freeze your query plans before running Tune Table. I HIGHLY recommend this practice, as it will allow you to run Tune Table and only take advantage of the new plan if it helps (you can test your query with %NOFPLAN to see if the new plans will help you).

Another note on Frozen Query Plans – due to this technology you can run Tune Table without effecting any query that you have already run on your system. It is a way for you to run Tune Table without worrying about effecting your current performance! NO EXCUSES! If you want to learn more about Frozen Query Plans you can view a Webinar I did on them here:

https://learning.intersystems.com/course/view.php?id=969

In closing, when it comes to Query Performance, you want to be cognizant of your Tune Table statistics. Tune Table gives our query optimizer the information it needs to make good decisions. Assuming you ’ ve already added indexes to your tables, then running Tune Table is the next step in making your queries blazing fast.

A special shoutout to Aaron Bentley for showing me this SQL formatter that I used for this article:
http://dpriver.com/pp/sqlformat.htm

And a second shoutout to Brendan Bannon who makes my articles sound actually comprehensible.

#Best Practices #Performance #SQL #Caché #InterSystems IRIS