Efficiently Handling Creating Blocks

You known what they say? Don’t believe everything you read on the internet. Creating blocks with DATACOPY is only the ***second slowest*** way you can do it. Do you want to know how to create a calculation using a faster performing technique to create blocks while calculating the members in those blocks?

Introduction

I see hundreds of examples of how to deal with block creation. It almost always is an issue when I am asked to get involved in projects that have performance issues. You will never hear me say “best practice.” There are reasons we all do things that go against what Oracle will say is a best practice. We have larger block sizes to improve reporting performance and we know it causes slower calcs. If the calc runs at night nobody cares if it runs 10 minutes or 60 minutes. We go against the grain on dimension order when data is sparse and performance on calculations is improved by ignoring the “ideal” order.

Block creation can be a bear if you don’t understand what blocks are or don’t understand how they get created. There are a bunch of ways to do it and there are many options that can be used. I want to educate on what a block is and show how to handle it the most efficient way because 9 out of 10 people don’t know how to handle it efficiently. I don’t know why, because it is easy to figure out if you think about it and understand some basic concepts, and even easier to implement it.

If you have a better way to do it share your techniques. Hopefully I don't say anything incorrectly and I shed some light on block creation for the majority of the people reading this.

What Is Block Creation

Essbase stores data in blocks. A block is the combination of dense dimensions. There is a block for every stored combination of sparse members. If you don't know how blocks work, I highly recommend you read Sparse, Dense, and Blocks for Dummies. If a block doesn't exist, it will only get created if you tell Essbase to create it. I will get into how to do this shortly.

When blocks don't exist, a lot of calculations appear to work intermittently. If you have this issue, it is likely due to the fact that it works when blocks already exist and doesn't when they don't exist. You can easily tell by loading a number to one combination of the dense members and run the calc again. Submitting a value will create the block if it doesn't exist. If it calc works after you submit some data, it is most likely the fact that the block was not there and is now.

Block creation is something you will likely encounter on every project. How do deal with it is what I want to tackle.

Using DATACOPY – What Most Teach You

Most people create blocks by using DATACOPY. I just read a post on LinkedIn that sparked my interest in writing this because I find it terribly frustrating when people/companies represent themselves as experts and teach people inefficient ways to do things. I have most certainly done this in my life in an effort to help people, but this topic is so basic and polarizing for me because it is SO EASY to do it right. By

right, I mean less code, faster execution, and easier to write in the first place. Here is likely what you have heard.

In this example, marketing expenses needs to be calculated as 5% over actuals. The method suggested is to copy data from Actual to Budget, which duplicates every Actual block for Budget. Then you wipe out the data in Budget. Then you set the data with your calculation to make Budget 5% more than Actual. It looks something like, excluding the FIX statements.

```
DATACOPY "Actual" to "Budget";
"Budget" = #Missing;
"Budget" = "Actual" * 1.05;
```

To reiterate, if you would run a calculation on Budget and no blocks existed, it would likely do nothing. If the calculation FIXes on Budget, and no blocks exist for Budget, it does nothing. If it already has some of the needed blocks, the calculation will appear to work on some of the data but not on others. To get around this, people will copy all the data from Actuals to Budget so all the needed blocks exist. Then they will clear the data from Budget. Then they will loop through Budget and run the logic. The result is this.

```
FIX([your fix statement], "Marketing Expenses")
  DATACOPY "Actual" to "Budget";
ENDFIX
FIX([your fix statement], "Actual")
  "Marketing Expenses" = #Missing;
ENDFIX
```

Does this work? Absolutely. Is it efficient? Not even close. A significant reason for calculation performance is looping through the same block more than once. Sometimes it is necessary, but think of it this way. If you had 50 Excel workbooks and you had to update all 50 of them. Would you open each one and edit one cell, then open them again and update another cell? No. You would open it and update all the cells, close it, and never open it again. You want your calculation to perform the same way. You want it to open a block, edit

everything in that block, and never open it again. I understand that is not always possible, so I am not suggesting this is a hard and fast rule.

This calculation, as it is, runs through the blocks 3 times. Each fix represents a and is very inefficient. It also creates blocks regardless of whether Marketing Expenses has data or not.

Don’t use this method!

Using CREATEBLOCKONEQ – Hopefully Nobody Teaches You

Another method is to use SET CREATEBLOCKONEQ in your FIX statement. Using this means we don’t have to use DATACOPY and we don’t have to set the result to null using #Missing. It makes the calculation smaller and would look like this.

```
FIX([your fix statement], "Budget")
  SET CREATEBLOCKONEQ ON;
   "Marketing Expenses" = "Marketing Expenses"->"Actual" *
1.05;
  SET CREATEBLOCKONEQ OFF;
ENDFIX
```

This will work for you. It will create the blocks as they are needed. But, it will likely take longer than the original example. Why? Without this setting, the calculation will go through every EXISTING block. It will calculate any block that exists, but will skip the blocks that don’t exist and you will get inconsistent results.

When we set CREATEBLOCKONEQ to ON, it will check every possible block and if it needs to be created it will. The problem with this is that unless your cube is freakishly dense, it will do a lot more than it needs to. Think of putting your car in 1st gear and driving on the interstate for

10 miles. You will go extremely slow and use an enormous amount of gas compared to driving it 5th gear, in which you would get there faster and use very little gas. That is what CREATEBLOCKONEQ does.

Look at your cube statistics and check out the difference between the number of blocks and the number of possible blocks. This runs on all the possible blocks and is likely millions, if not billions of blocks that have no data.

Since the performance on this is really bad, it is rarely used. That said, if you need to run a calculation on a specific combination, like 1 block, or 10 blocks, using this is an easy way to minimize your effort and the additional time by using this is not material, and the blocks have to be created anyway. This would be like putting the car in 1st gear and driving up your driveway. Not super- efficient, but it is only 5 seconds, so who cares.

An Alternate Approach Nobody Talks About

First, will this work in every situation? No. Will it work in 99.9% of them? Yep! This example assumed Scenario is a sparse dimension.

We have established that if we fix on blocks that don’t exist, nothing will happen. Is it necessary that we create the blocks first, then calculate them? Absolutely not. We can create the block by calculating it.

Blocks will get created IF the left side of the equation is a sparse member. If you must have a cross dim operation, and the left most member of the cross dim is sparse, it will create the block. In the original example, we wanted to grow Marketing Expenses 5% over the prior year. If we fix on Actual, where all the blocks that we need grow the expense

exist, then we set Budget to 105% of Actual, the block will get created. All of the blocks will get created if they don't already exist. If there is no block for Actual, then Budget will be 0 anyway, so we don't have to worry about if the right blocks exist for Actual.

```
FIX([your fix statement], "Actual", "Marketing Expenses")
  "Budget" = "Marketing Expenses" * 1.05;
ENDFIX
```

If you think about this, we are running through all the Actual blocks, and then we set the Budget to a 5% more than Actual. Rather than fixing on the calculations, or destination, and calculating the destination, we fix on the source, where the blocks exist, and set the destination to a value. Since we have a sparse member on the left side of the equation, and we are FIXing on Actual, where all the blocks are, we will NOT be skipping anything and you should get a total Budget of 5% more than Actual.

Will this create too many blocks? Possibly. We may not have Marketing Expenses in all the existing Actual blocks. We can optimize this by adding a IF to check to see if Marketing Expenses is either a 0 or #Missing. Rather than checking for both, I just add a 0 to it and then check for 0. 0 + #Missing is 0 and it is just a little more efficient to process and write.

```
FIX([your fix statement], "Actual")
  "Marketing Expenses"(
  IF("Marketing Expenses" + 0 <> 0)
     "Budget"->"Marketing Expenses" = "Marketing Expenses" *
1.05;
  )
  ENDFIX
ENDFIX
```

To summarize, we

- FIX on Actual

- Only calculate Budget if Marketing Expense is not zero and not #Missing
- Have a sparse member on the left side of the equation
- Have a sparse member on the left most side of the cross dimensional operator This will likely result in less blocks created as it only creates blocks where Marketing Expenses is #Missing or 0 but will create and calculate any block need to be created at the same time. What is the benefit of this? The calculation is easier to write, it takes less time to write, and it is much more efficient to execute.

Results Of Real-World A Dataset f The calculations are plain Jane. I am not using threading or any other settings to keep it simple and ensure I am comparing apples to apples.

The database I have has more data in the Plan scenario so I am replicating the same logic but using different scenarios. I need set the BlockCreationTest scenario to 5% more than the OEP_Plan scenario, rather than the prior example of setting Budget to 5% more than Actual. Different scenarios but the exact same concept.

DATACOPY Method

The calculation to copy data to create blocks, set destination to missing, then calculate the growth FIXing on the destination, will look like this.

```
FIX("FY18", "OEP_Working", "USD", "c01",
    @Relative("Source", 0),
    @Relative("Total_Department", 0),
    @Relative("ComputerTech", 0))
  DATACOPY "OEP_Plan" TO "BlockCreationTest";
ENDFIX
FIX("FY18", "OEP_Working", "USD", "c01",
    @Relative("Source", 0),
    @Relative("Total_Department", 0),
    @Relative("ComputerTech", 0),
```

```
    "BlockCreationTest")
  "Regular_Cases" = #Missing;
ENDFIX
FIX("FY18", "OEP_Working", "USD", "c01",
    @Relative("Source", 0),
    @Relative("Total_Department", 0),
    @Relative("ComputerTech", 0),
    "BlockCreationTest")
  "Regular_Cases" = "Regular_Cases"->"OEP_Plan" * 1.05;
ENDFIX
```

The data copy took 41 seconds. Updating the destination to #Missing took 5 seconds. The calculation took 42 seconds. The total time was 88 seconds and it created 66,724 blocks. This went through the blocks 3 times.

FIX On Source And Calculate Destination Method

This time, I just FIXed on the source, and used the blocks on the source to create and calculate the destination. The calculation looks like this.

```
FIX("FY18", "OEP_Working", "USD", "c01",
    @Relative("Source", 0),
    @Relative("Total_Department", 0),
    @Relative("ComputerTech", 0),
    "OEP_Plan")
  "Regular_Cases"(
  IF( "Regular_Cases" + 0 <> 0)
     "BlockCreationTest"->"Regular_Cases" = "Regular_Cases" *
1.05;
  ENDIF
  )
ENDFIX
```

This created slightly fewer blocks in 63,739. The difference means that Regular Cases didn’t have a value in all the existing source blocks. The calculation took 6 seconds. That is an improvement of over 8,000%! This method created the

necessary blocks and calculated the correct values in 6 seconds compared to 89 seconds using the other method. It went through the blocks one time.

Optimized DATACOPY

Just to be fair, I optimized the calculations for the DATACOPY methodology. The example provided in the article that prompted me to write this was inefficient. I wanted to squash any comments that would suggest the DATACOPY is just as fast if it was written efficiently.

```
FIX("FY18", "OEP_Working", "USD", "c01",
    @Relative("Source", 0),
    @Relative("Total_Department", 0),
    @Relative("ComputerTech", 0),
    "Regular_Cases")
  DATACOPY "OEP_Plan" TO "BlockCreationTest";
ENDFIX
FIX("FY18", "OEP_Working", "USD", "c01",
    @Relative("Source", 0),
    @Relative("Total_Department", 0),
    @Relative("ComputerTech", 0),
    "BlockCreationTest")
  "Regular_Cases" = #Missing;
ENDFIX
FIX("FY18", "OEP_Working", "USD", "c01",
    @Relative("Source", 0),
    @Relative("Total_Department", 0),
    @Relative("ComputerTech", 0),
    "BlockCreationTest")
  "Regular_Cases"(
  IF( "Regular_Cases"->"OEP_Plan" + 0 <> 0)
    "Regular_Cases" = "Regular_Cases"->"OEP_Plan" * 1.05;
  ENDIF
)
ENDFIX
```

The results were better. The data copy took 6 seconds. Updating the destination to #Missing stayed the same and completed in 5 seconds. The calculation of BlockCreationTest

took 7 seconds.

That is still 18 seconds, which is 2.5 times slower than FIXing on the source and calculating the destination. I don't know why you would ever need the step that sets everything to #Missing because it would get overwritten with the third step, but would be 0 anyway. Even if that step gets removed, this method is still twice as fast.

CREATEBLOCKONEQ ON Method

Before I get into the results, take a look at these statistics. The FIX statement runs on

- 13 Sources
- 23 Departments
- 2,382 Companies
- 25,001 Products

13 x 23 x 2,382 x 25,001 is 17,806,162,218 if my math is correct. That is possible blocks that can exist. If you remember, the calculations above created 66,724 and 63,739 blocks for the one without the if and with the if, respectively. If we extrapolate out the results for a calculation that took 6 seconds that iterated through 66,724 blocks (we know this because it created a block for every source block that existed), to run through 17 billion blocks, it will take an estimated 500 hours! Remember, turning this on will go through ever possible block and if it needs to be created it will create it. If not, it won't. The calculation is as follows.

```
FIX("FY18", "Regular_Cases", "USD", "c01",
    @Relative("Source", 0),
    @Relative("Total_Department", 0),
    @Relative("ComputerTech", 0),
    "BlockCreationTest")
  SET CREATEBLOCKONEQ ON;
  "OEP_Working" = "OEP_Plan" * 1.05;
```

```
  SET CREATEBLOCKONEQ OFF;
ENDFIX
```

The calc had to be adjusted slightly because even with this setting on, it only creates the block if a sparse member is used on the left side of the equation. I moved OEP_Working from the FIX to the left side of the equation and moved BlockCreationTest to the FIX. I did this in good faith because there is ONLY data in Working, but there is data in multiple Scenarios, so this should run faster than if I left it the way it was in the other calculations.

I stopped this calculation after 5 hours and it only created 223 blocks at that time. If I extrapolate that out, it would take 1,500 hours to finish. Even though I only need to create 66k blocks, I have to go through 17B to figure out which ones need to be created, verses fixing on the source and it only running through 66K blocks.

To The Doubters

I get feedback all the time that this method isn’t possible all the time. There are times when it isn’t possible, but in 30 years of doing this, I can think of maybe 5 times I had to work around it. If you want to allocation a number based on history, you fix on the history and set the destination equal to the history * a cross dim to your rate. If you need to allocation based on percentages entered, then you fix on where the percentages are entered and set the destination to the correct value, as the blocks are created when the rate is entered. This doesn’t just apply to scenario to scenario block creation. You may enter a rate at your entity for eliminations, but not at the product level. You still are only going to allocate down to the products that have budget, or history, so you still can fix on where the products have data and use the rate at no product to calculate your numbers.

If you are allocating the data or have a driver that requires

calculations to be created, it has to have some driver somewhere that exists at the level you want to allocate FROM, and if you have that, you can use this method.

I Accept Your Challenge

If you have a situation where you are having challenges with this logic and think you have to use DATACOPY, challenge me to come up with a way to do it. I don't want you to EVER have to use DATACOPY!

A Cautionary Tale

With all this said, is there a drawback? Yes and no. This will always create the needed blocks. Keep in mind when you put a sparse member on the left-hand side and if you don't have your FIX isolated to only what you need to calculate, you can potentially create a lot more blocks than you want. You will NEVER create a block at every possible block with this method because if you FIX on something, it ONLY calculates where blocks exist.

If you ran the following calculation on an empty database, nothing would happen because there are no blocks that it would execute the calculation on. To prove this, I cleared all the blocks in the Scenario BlockCreationTest and ran the following calculations.

```
FIX("FY18", "OEP_Working", "USD", "c01",
    @Relative("Source", 0),
    @Relative("Total_Department", 0),
    @Relative("ComputerTech", 0),
    "BlockCreationTest")
  "Regular_Cases" = 1;
ENDFIX
```

This ran in under a second and created no blocks because BlockCreationTest has no blocks.

What Hasn’t Been Discussed

There are two other ways to create blocks.

CREATEBLOCK

First is the @CREATEBLOCK. You can pass a list of members to this to create blocks. I have used this in some situations where I needed to walk balanced to the next year and I wanted to make sure the next year existed. This is just one example. It did add time to the calculation, but wasn’t significant for small datasets. A couple things to keep in mind.

1. You have to be extremely careful with this because you can blow up your database if used incorrectly.
2. If you use this within a fix, you only have to pass the member that is different than the block you are on.
3. In most situations, this isn’t necessary because of the preferred method above.
4. It will, in most situations, still require an extra pass of the blocks and negatively impact the calculation speed.

Here is an example.

```
FIX("FY18", "OEP_Working", "USD", "c01",
    @Relative("Source", 0),
    @Relative("Total_Department", 0),
    @Relative("ComputerTech", 0),
    "OEP_Plan")
  "Regular_Cases"(
  IF( "Regular_Cases" + 0 <> 0)
    @CREATEBLOCK("BlockCreationTest");
  ENDIF
  )
ENDFIX
```

Comparing this to the other way to create blocks, this took 40 seconds and created the correct blocks. If you remove the if, however, it will create every possible block combination

possible, which is why you have to be extremely careful with this method.

Groovy

The other option is to use Groovy to create the blocks. It would be safer, but probably slower than the @CREATEBLOCK method and more complicated to write and I didn’t even bother to test it.

My Hope

I hope you read this, understand it, and have a bullet proof way to deal with block creation and a more efficient way than you may have ever thought of or been taught. I feel so strongly about this that I am more than happy to have a quick conversation if you are finding it difficult to use it.