Make a Good Burger – A Corticon Solution – by Mike Parish

Problem Definition

As the owner of a fast food restaurant with declining sales, you know that your customers are looking for something new and exciting on the menu. Your market research indicates that they want a burger that is loaded with everything as long as it meets certain health requirements. Money is no object to them. The ingredient list in the table below shows what is available to include on the burger:

Item	Sodium(mg)	Fat(g)	Calories	Item cost($)
Beef Patty	50	17	220	$0.25
Bun	330	9	260	$0.15
Cheese	310	6	70	$0.10
Onions	1	2	10	$0.09
Pickles	260	0	5	$0.03
Lettuce	3	0	4	$0.04
Ketchup	160	0	20	$0.02
Tomato	3	0	9	$0.04

You must include at least one of each item and no more than five of each item. You must use whole items (for example, no half servings of cheese). The final burger must contain less than 3000 mg of sodium, less than 150 grams of fat, and less than 3000 calories. To maintain certain taste quality standards you'll need to keep the servings of ketchup and lettuce the same. Also, you'll need to keep the servings of pickles and tomatoes the same.

Question: *Offer several recipes for a good burger. What is the most and the least expensive burger you can make?*

Observation

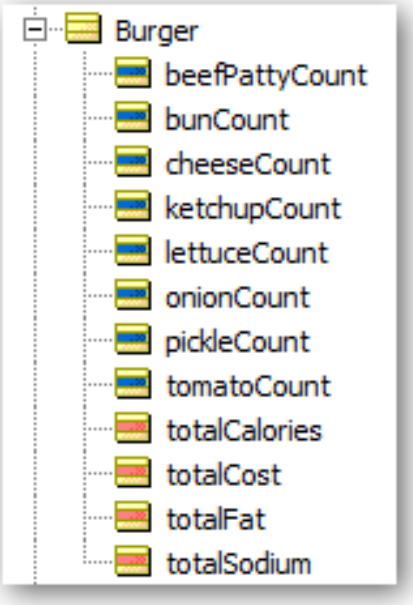
Although this is not typical of the kinds of rules that we see in business it can still be modeled using Corticon decision tables and the model can be executed to produce a solution. It falls into the "Generate and Test" category of solutions for solving constraint based problems.

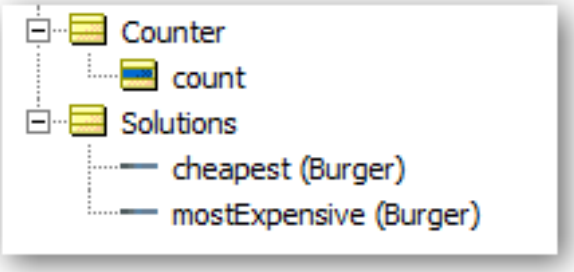
The Vocabulary

The **Burger** entity tracks the composition of a burger with counters for each of its ingredients.
These are shown with blue decorations.
The total calories, sodium, fat and cost which will be calculated are shown in orange.

The rules will create many instances of a burger according to the health requirements specified in the problem description.

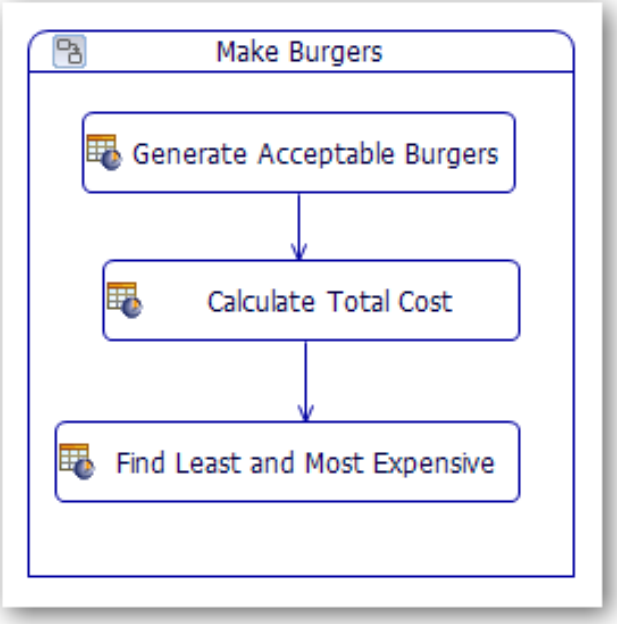
From these we will select the least and most expensive.





The **Counter** is used to indicate how many times we are allowed to add each ingredient.
The **Solutions** entity is used to track the most and least expensive burger.
We'll use 1:1 associations to categorize the burgers in the solution entity.

The Decision Structure



The problem is solved in three steps.

Step 1 generates all possible burgers.
Of these, about 5000 meet the health and other restrictions (i.e. "healthy" burgers)

Step 2 then calculates the cost of each "healthy" burger.

Step 3 finds the most and least expensive.

Clearly the least expensive burger is the one that has just one of each item.
We could make the problem more interesting by also requiring a minimum sodium, fat and calorie count in addition to a maximum to ensure the burger is tasty and filling.

Natural Language View of the Rules

There are just three rule sheets required

Generate Acceptable Burgers

This rule sheet will examine all possible burgers and select only those that meet the restrictions specified in the problem definition.

Conditions	1
What is the total sodium in this burger?	< 3000
What is the total fat in this burger?	< 150
What is the total calories in this burger?	< 3000
Actions	
Post Message(s)	
This burger is acceptable.	✓

Calculate the Total Cost

This is done by summing the products of the number of items and the contribution of each item to the total cost, sodium, fat and calorie count

Conditions	0
Actions	
Post Message(s)	
Calculate the total cost of each "healthy" burger	✓
Calculate the total sodium in each burger	✓
Calculate the total fat	✓
Calculate the total calories	✓

Find the Least and Most Expensive

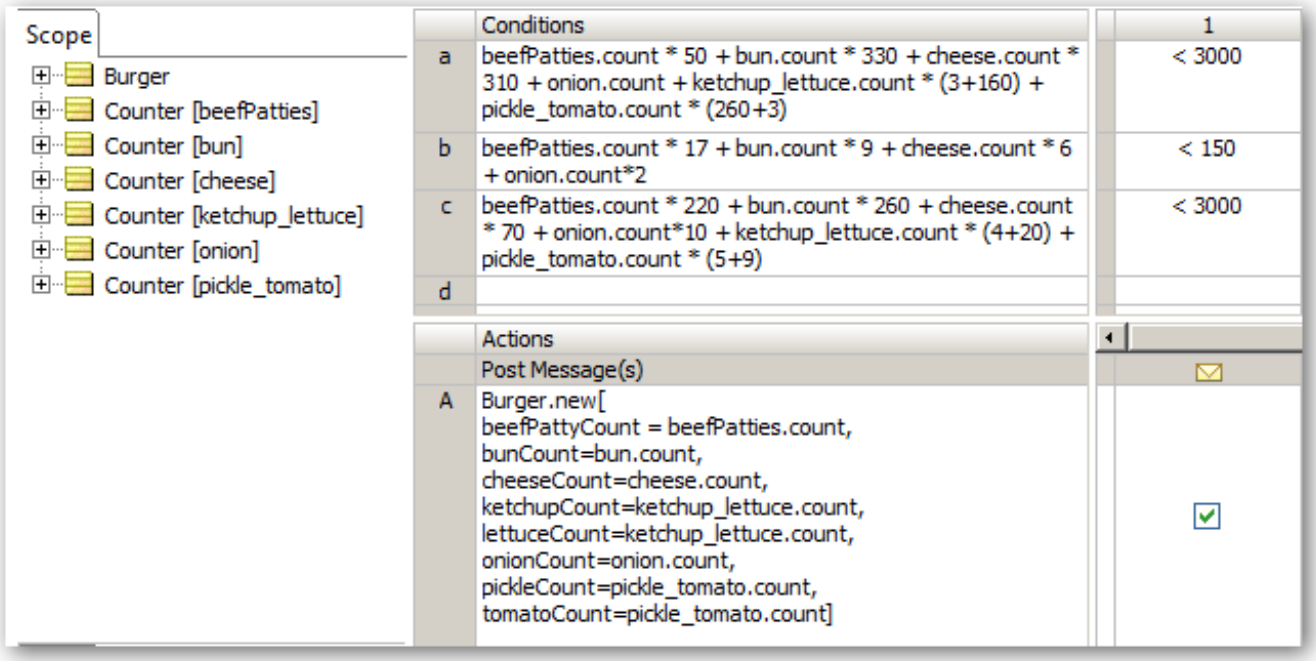
This can be done by sorting the burgers in descending or ascending order of cost and selecting the first.

Conditions	0
Actions	
Post Message(s)	
Find the most expensive burger	✓
Find the least expensive burger	✓

Implementation Details

To understand how the solution works we need to look at the underlying expressions that implement the natural language view.

Step 1 Generate "Healthy" Burgers



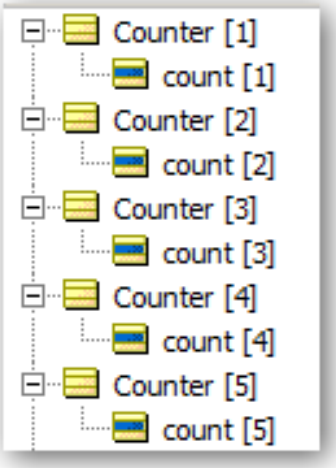
Scope

- Burger
- Counter [beefPatties]
- Counter [bun]
- Counter [cheese]
- Counter [ketchup_lettuce]
- Counter [onion]
- Counter [pickle_tomato]

	Conditions	1
a	beefPatties.count * 50 + bun.count * 330 + cheese.count * 310 + onion.count + ketchup_lettuce.count * (3+160) + pickle_tomato.count * (260+3)	< 3000
b	beefPatties.count * 17 + bun.count * 9 + cheese.count * 6 + onion.count*2	< 150
c	beefPatties.count * 220 + bun.count * 260 + cheese.count * 70 + onion.count*10 + ketchup_lettuce.count * (4+20) + pickle_tomato.count * (5+9)	< 3000
d		

	Actions	
	Post Message(s)	
A	Burger.new[beefPattyCount = beefPatties.count, bunCount=bun.count, cheeseCount=cheese.count, ketchupCount=ketchup_lettuce.count, lettuceCount=ketchup_lettuce.count, onionCount=onion.count, pickleCount=pickle_tomato.count, tomatoCount=pickle_tomato.count]	✓

The key to solving this problem is the use of the entity called Counter which has just one attribute called count. We create several aliases to Counter called beefPatties, bun, cheese etc. corresponding to each ingredient.



Corticon then naturally will generate ALL POSSIBLE COMBINATIONS (15625 of them) based on the values we supply in the input data. Eg:

Based on the calculations of sodium, fat and calories, rule 1 will only generate a new burger (5192 of them) if it meets the health restrictions.

Notice that by using a single alias for ketchup_lettuce and another for pickle_tomato we can ensure that the number of ketchup and lettuce are the same and that the number of pickles and tomatoes are the same.

Step 2 Calculate Totals

We can now calculate the total cost of the generated burgers using the individual costs in the problem definition.

Sodium, fat and calories would be implemented in a similar fashion.

Conditions	0

Actions	
Post Message(s)	
thisBurger.totalCost=	
thisBurger.beefPattyCount * 0.25+	
thisBurger.bunCount * 0.15+	
thisBurger.cheeseCount * 0.10+	
thisBurger.onionCount * 0.09 +	✓
thisBurger.ketchupCount * 0.02 +	
thisBurger.lettuceCount * 0.04 +	
thisBurger.pickleCount * 0.03 +	
thisBurger.tomatoCount * 0.04	

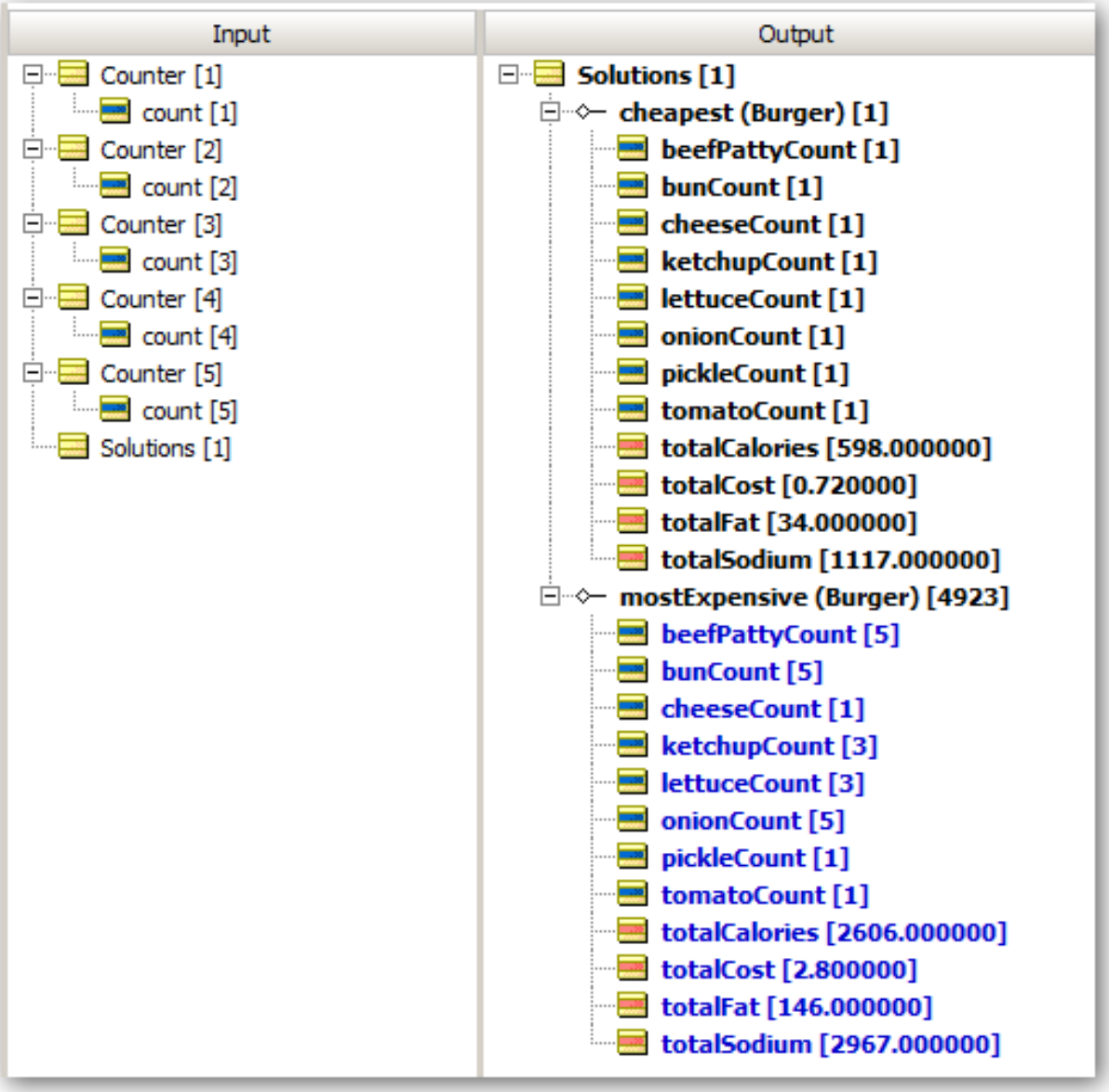
Step 3 Determine the Least and Most Expensive Burgers

Finally we can sort and select the least and most expensive.

Solutions.mostExpensive=burgers->sortedByDesc(totalCost).first	✓
Solutions.cheapest=burgers->sortedBy(totalCost).first	✓

That’s it – that’s all there is to the Corticon solution!!!

Test Case



Summary

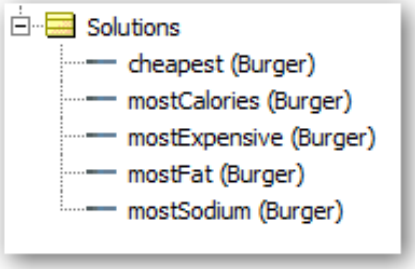
You can see how a fairly complex problem can be solved quite easily and elegantly in Corticon without resorting to complex procedural programming or looping by taking advantage of Corticon's built in ability to use the SCOPE to consider all possibilities. Instead of trying to "solve" the problem procedurally we simply state the conditions for a solution and allow Corticon to find it.

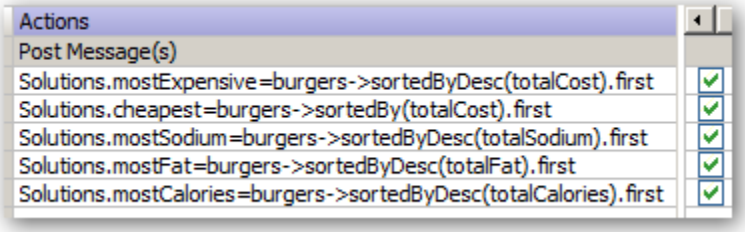
The rest of this article discusses various enhancements that can easily be made to the basic rule model.

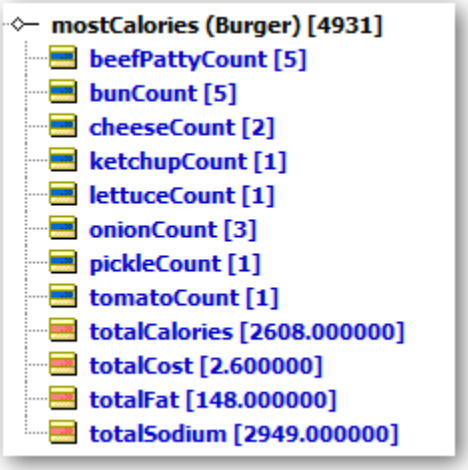
Extending the Solution

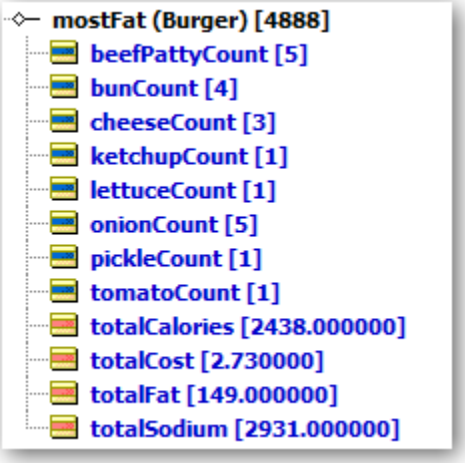
Enhancement #1

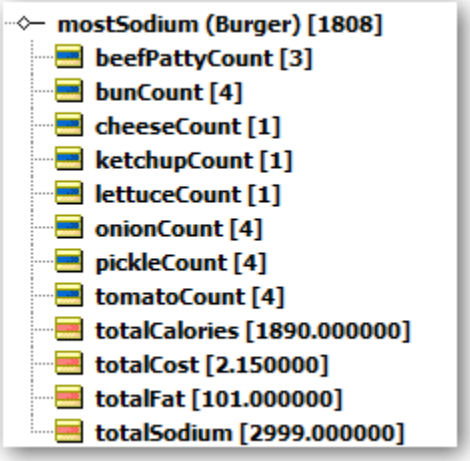
We can very easily add logic to find the burger with the most sodium, fat or calories that is still considered “healthy”.





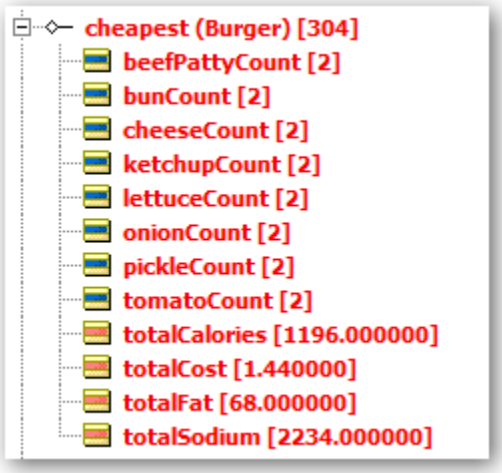


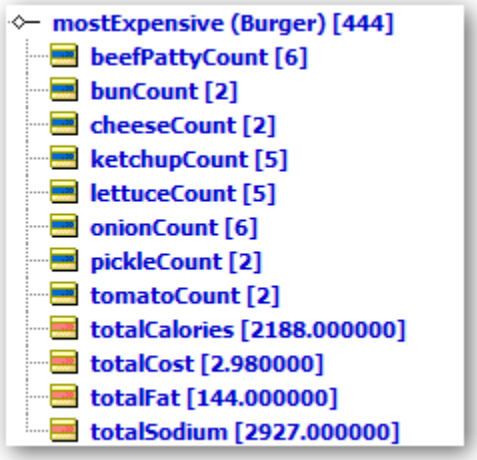




Enhancement #2

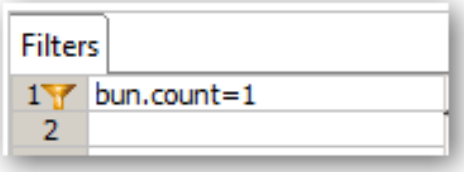
What happens if each ingredient must be present 2-6 times?
We simply change the input data values to list 2-6 as the possible item counts
We'll find there are now only 444 "healthy" burgers"



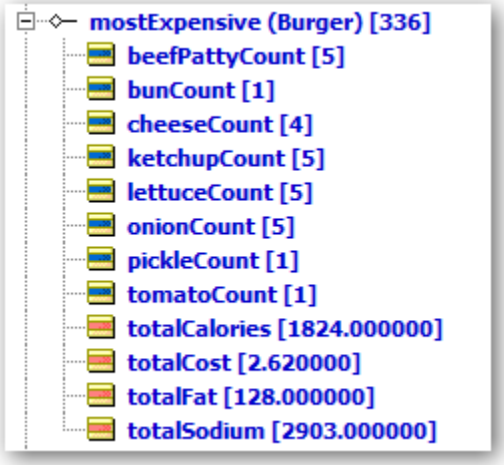


Enhancement #3

Suppose we wanted to run the original problem but with the bun count fixed at 1. This can easily be done by adding a filter to the "Generate Burgers" rule sheet:

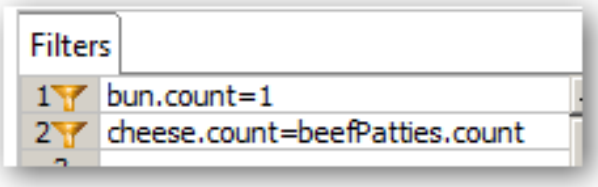


Now the most expensive burger is this one:

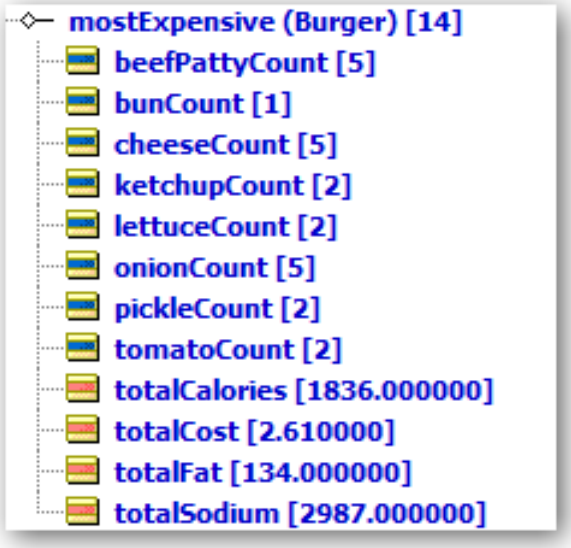


Out of 2116 possible burgers.

If, additionally, we wanted to ensure that the cheese count was the same as the patty count we could add this filter



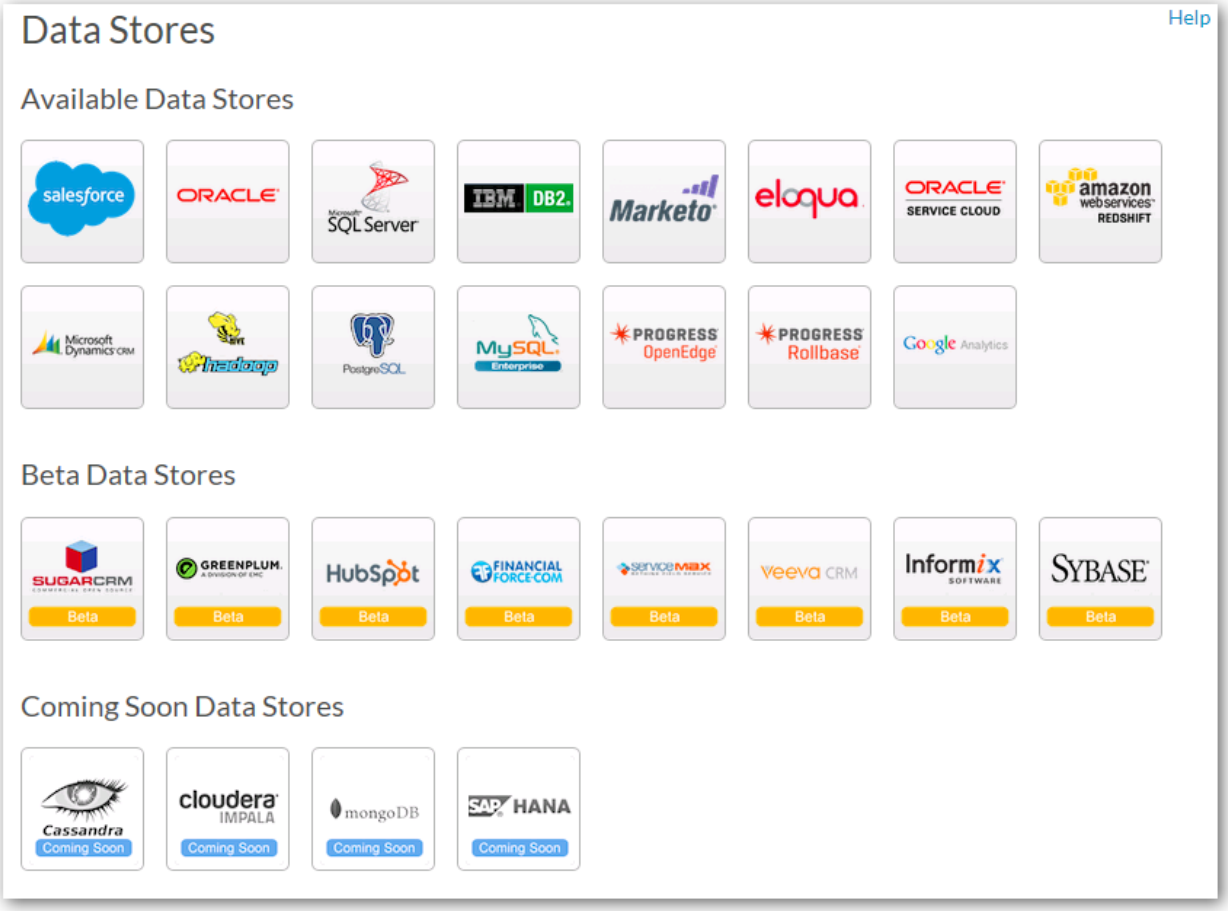
To get this result (just one cent less expensive)



Out of 415 possible burgers

Generalizing the Solution

The thresholds and ingredient content for sodium, fat, calories and cost could be read from an external source to further generalize the solutions. Corticon's Enterprise Data Connector provides a way to do this without the need to write SQL queries or to understand databases.
Corticon can connect to many different data sources:



Regardless of the data source, setting things up in Corticon is the same.
If we had these tables in the database:

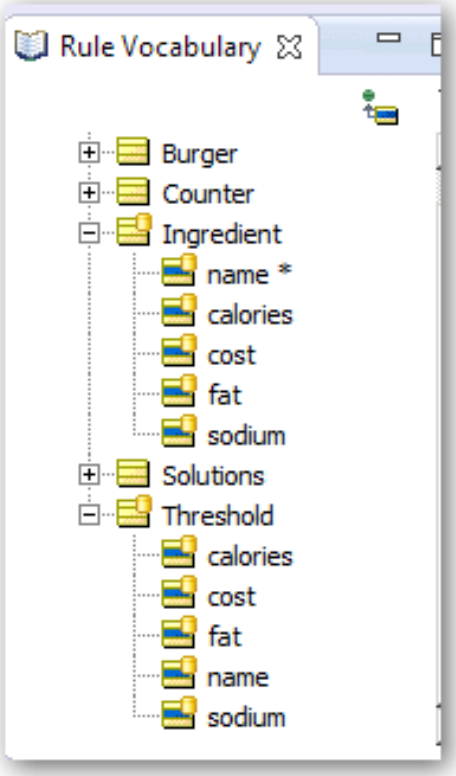
Ingredients

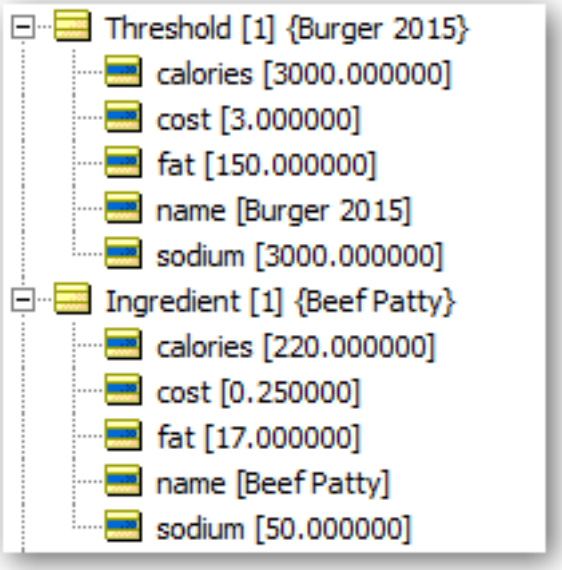
name	calories	cost	fat	sodium
Beef Patty	220.000000	0.250000	17.000000	50.000000
Bun	260.000000	0.150000	9.000000	330.000000
Cheese	70.000000	0.100000	6.000000	310.000000
Ketchup	20.000000	0.020000	0.000000	160.000000
Lettuce	4.000000	0.040000	0.000000	3.000000
Onion	10.000000	0.090000	2.000000	1.000000
Pickles	5.000000	0.030000	0.000000	260.000000
Tomato	9.000000	0.040000	0.000000	3.000000

Thresholds

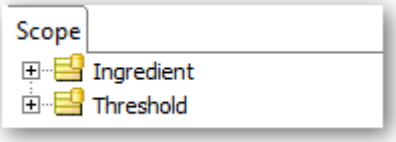
name	calories	cost	fat	sodium
Burger 2015	3000.000000	3.000000	150.000000	3000.000000

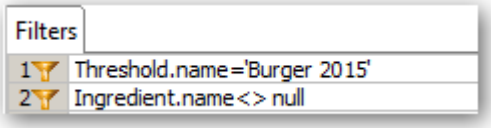
Then the vocabulary in Corticon might look like this:



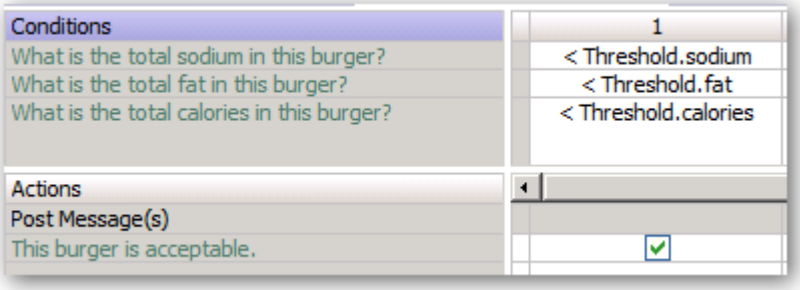


In order to read these records from the database it's just a matter of flagging the objects in the scope section as "**Extend to Database**":

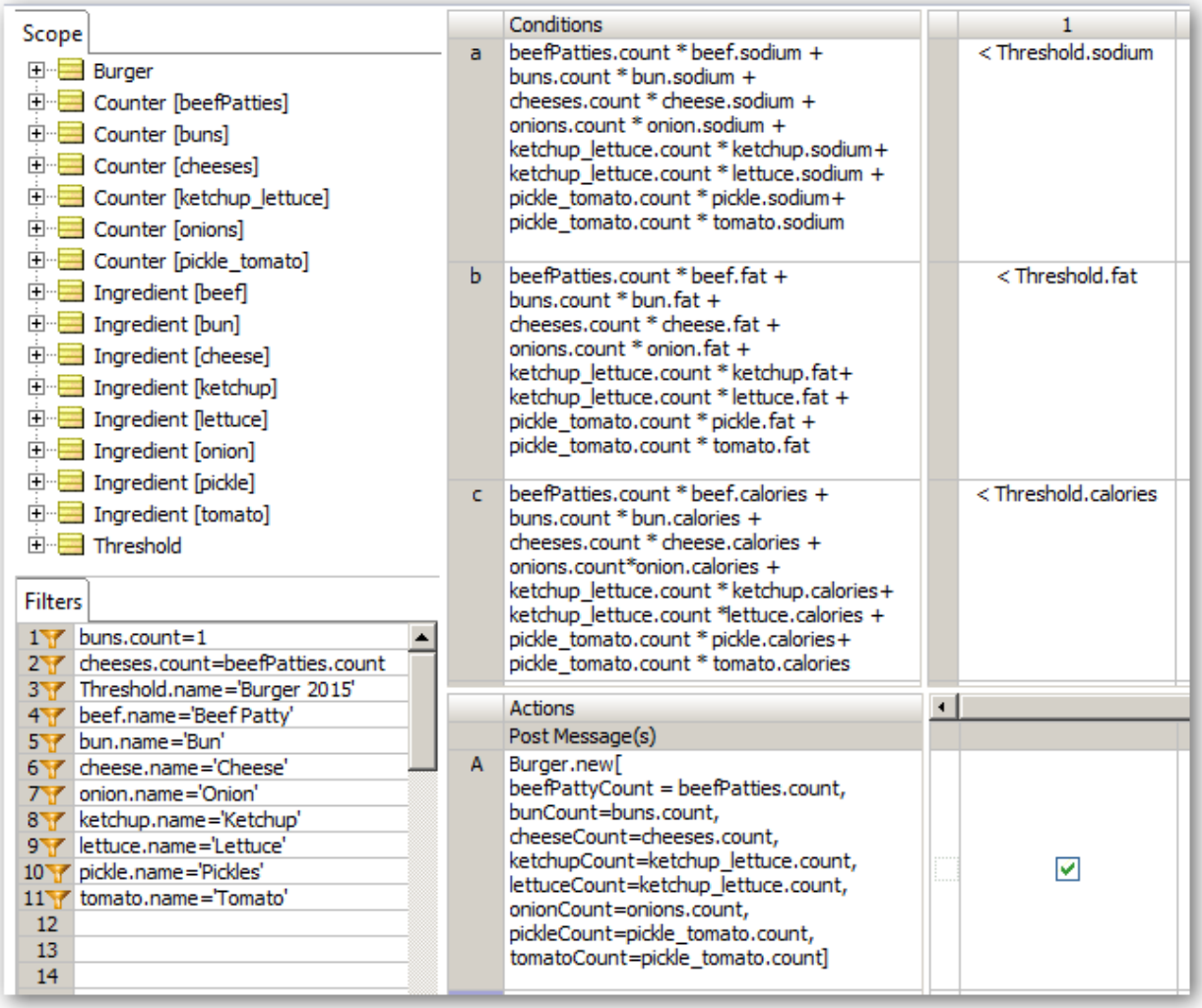




Adding a filter allows us to select the appropriate thresholds for 2015.
Then in the rule sheets we can refer to the threshold values read from the database:

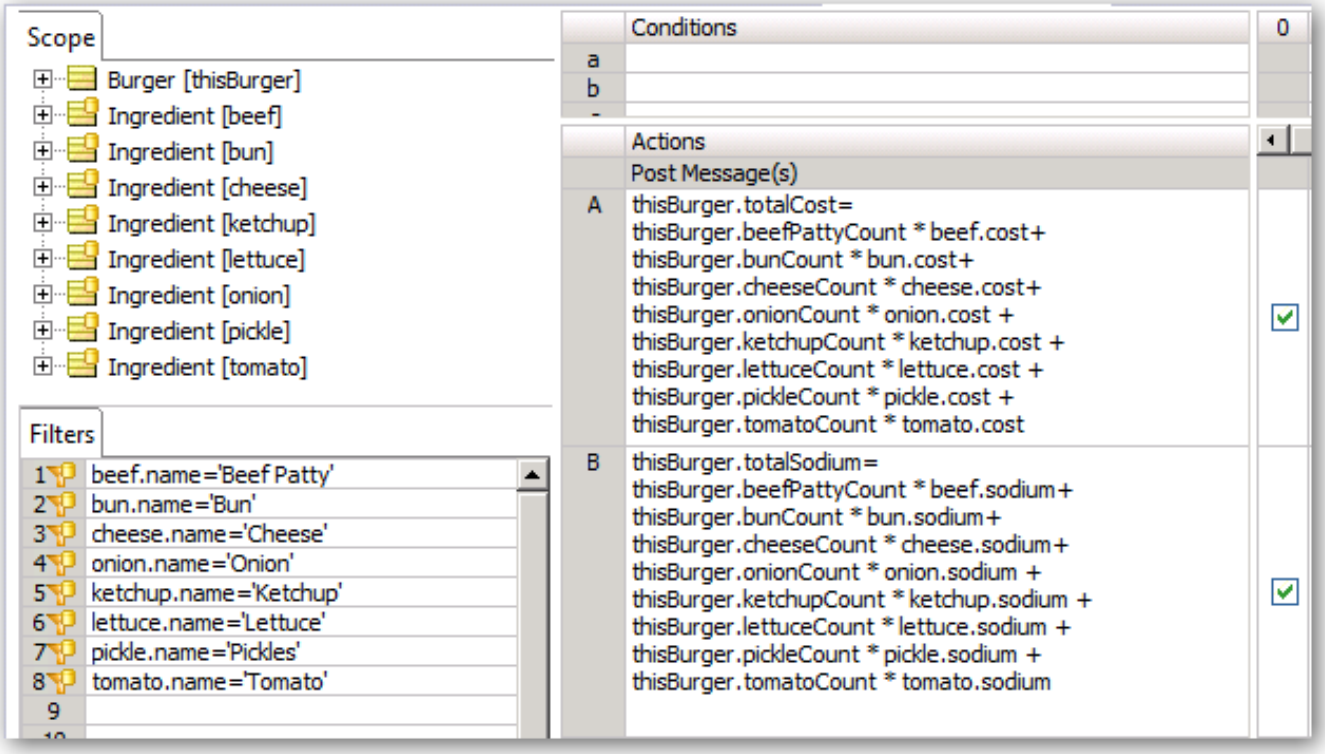


The "Generate Burgers" rule sheet implementation will look like this:



Again we use aliases to obtain the correct values to apply in the calculations.
Even the count of items could be stored in the database and we could allow different ingredients to have different ranges of counts.

The calculation rule sheet will look like this



Notice how we use a combination of aliases and filters to provide more specific references (such as lettuce.cost) instead of the generic Ingredient.cost. The reference to thisBurger will apply across all burgers that were generated by the earlier rule sheet as “healthy”.
Corticon “knows” that it should “loop” when there is more than one burger. We don’t have to tell it that.

Using Corticon for Analytics

Once we have constructed “healthy” burgers we can add rule sheets to run various analytic queries against this burger database.

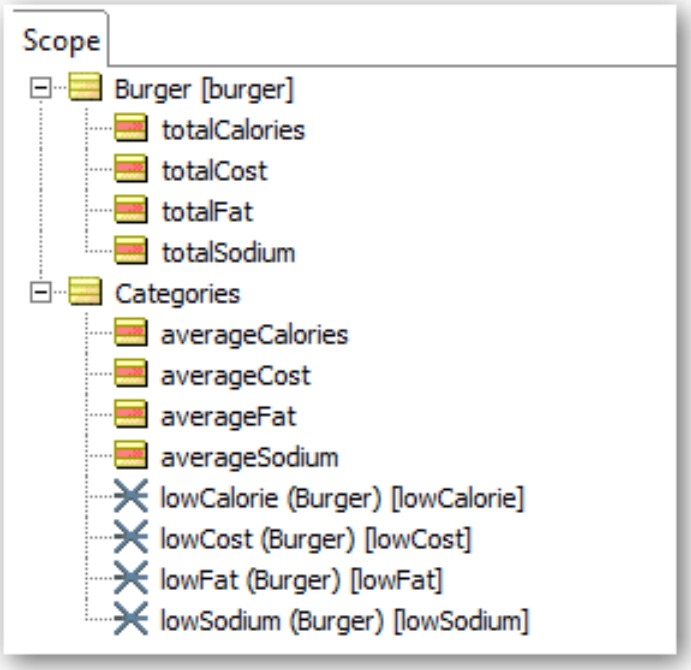
Conditions	0	1	2	3	4
Is the total cost less than half the average?		T	-	-	-
Is the total sodium less than half the average?		-	T	-	-
Is the total fat less than half the average?		-	-	T	-
Is the total calories less than half the average?		-	-	-	T
Actions					
Post Message(s)					
Calculate average cost	✓				
Calculate average sodium	✓				
Calculate average fat	✓				
Calculate average calories	✓				
Add this burger to the low cost category		✓			
Add this burger to the low sodium category			✓		
Add this burger to the low fat category				✓	
Add this burger to the low calorie category					✓

Which can be implemented like this:

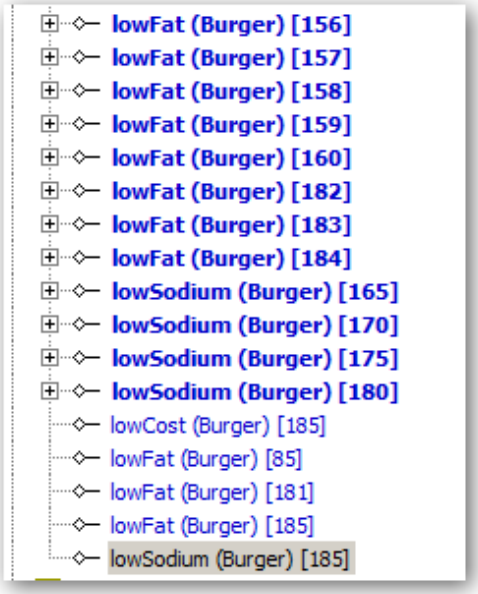
Conditions	0	1	2	3	4
burger.totalCost < 0.50* Categories.averageCost		T	-	-	-
burger.totalSodium < 0.50 * Categories.averageSodium		-	T	-	-
burger.totalFat < 0.50 * Categories.averageFat		-	-	T	-
burger.totalCalories < 0.50* Categories.averageCalories		-	-	-	T
Actions					
Post Message(s)					
lowCost += burger		✓			
lowSodium += burger			✓		
lowFat += burger				✓	
lowCalorie += burger					✓
Categories.averageCost = burger.totalCost->avg	✓				
Categories.averageSodium = burger.totalSodium->avg	✓				
Categories.averageFat = burger.totalFat->avg	✓				
Categories.averageCalories = burger.totalCalories->avg	✓				

Corticon automatically determines that the appropriate average needs to be calculated before the corresponding rule can be applied even though we have written the expressions at the bottom of the rule sheet.

This rule sheet uses the following scope to define the context:



Notice that we use many to many associations since a burger can belong to many categories at once. Results in the tester would look something like this:



Apparently there are no low calorie burgers!