Decision Management Community Challenge February 2021

Benchmark "Medical Services": High Performance with Large Decision Tables

(Bob Moore, JETset Business Consulting, 26 May 2021)

1 Problem Statement (from the web site)

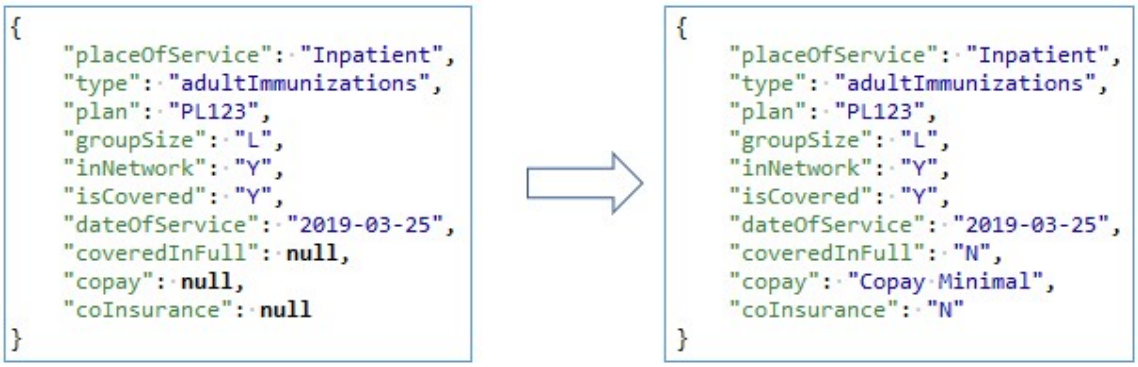
This challenge is intended to compare the **performance** of different decision service implementations. It has only one (but large) decision table and a set of JSON input structures. We expect that DM practitioners will implement this simple decision service using their favorite BR&DM tools and will provide *the actual execution time* of their solutions.

Problem. A company that successfully used SQL to access their DB-based medical services decided to migrate to the Serverless architecture. The first Decision Microservice they want to implement is called "Determine Medical Service Coverage". It should determine coverage attributes for various medical services using the decision table:

Conditions								Conclusions		
Place Of Service	Service Type	Plan	Group Size	In Network	Is Covered	Effective Period Start	Effective Period End	Covered in Full	Copay	Coinsurance

This table has **16,369 rules** for various combinations of medical service characteristics used in the Condition columns. Click here to see and save this decision table in CSV format.

Expected Solution. You need to create a RESTful decision service that takes a medical service in JSON format and fills out its coverage attributes like in this example:



We expect that you will download 10 JSON test cases and execute your service against them producing an output similar to this one. Please report the actual average service execution time for these 10 cases (without network overhead).

*We encourage the submitters to deploy their implementations on-cloud as a **RESTful decision microservice** and publish its endpoint URL and the exact JSON request. It would allow anybody to execute your solution from commonly used POSTMAN. Alternatively, you may explain how our readers could install and execute your decision model to check its actual performance.*

Please submit your solutions to DecisionManagementCommunity@gmail.com. A summary of all solutions will be placed in the following table:

.

2 Context

Since I have no connection with any vendors, and due a combination of events including Covid-19, I am for practical purposes retired, I wasn't particularly interested in this challenge, but Jacob to have a look at it, and I decided investing a little effort might be fun.

By the time I got started to do so, three solutions had been presented. However, I struggled a bit to make sense of how to compare the benchmarks, since none of them seemed to be measuring performance in quite the same way. It certainly was a surprise to see the highest performance from a Prolog[1] based solution. And the 'worst' performance was from the only solution which actually met up with the desire for publicly accessible RESTful decision microservice[2]. For various reasons, I put this aside for a while, and a number of additional solutions have been provided.

To compare like with like, one would need the same environment for benchmarking. Given I didn't have access to the tools used to compare like with like, it seemed reasonable to try and build a 'yardstick' implementation which I could then try deploying in similar environments so if solution A was twice as fast as the yardstick in one environment, and solution B three times as fast as the yardstick in a second environment, then one could reasonably say this implied solution B would be about 50% faster than solution A if both were in the same environment.

3 A Custom Solution

3.1 First Steps

Lacking access to a commercial tool, my first choice for a decision engine is generally RedHat's Drools – despite its obtuse documentation. However, having upgraded to Java 11, I found my existing installation of Drools no longer worked (Eclipse issues) and attempting to install the latest version of Drools, gave me all sorts of weird error message (Drools Eclipse plugin issues). So, I gave up! … but see section 4.

I've never delved into the mechanics of actually building a rule engine, but it always seemed to me that it should be pretty easy to build a 'simple' decision table engine. 'Simple' in this context broadly describes decision tables where the columns are either 'categorical' (i.e., strings) or 'numeric', the tests are all either 'equals' (for 'categorical') or the ordering comparisons for numeric' columns.

---

[1] Due to Matteo Redaelli see https://matteoredaelli.netlify.app/developer/dmcommunity_org_challenge-feb-2021/

[2] Due to Rob Parker see https://dmcommunity.files.wordpress.com/2021/02/challenge2021feb.robparker-2.pdf

Since I've implemented a few REST based web services using Java, I settled on this as my implementation technology.

In setting out to build a decision table engine, what I wanted was a solution which was independent of the actual content of the decision table. So, it should be able to consume any 'simple' table decision, regardless of the number of columns or rows. Since we were presented with the decision table as a CSV file, my idea was simply to consume the CSV file. In practise, the logic involved in doing this could equally easily be applied regardless of where the expression of the decision table was held, as a CVS file on some file store or at some designated URL, or was stored in a database. However, to make the solution easy to test out in a variety of environments, I found it most convenient to store it as a resource file to be packaged in the Java jar file.

3.2 Completing the Decision Table Specification

Having come this far, I then determined there was another problem to be faced. I had a CSV file and a JSON file (see section 1), but what was the relation between them? To a person, it is pretty obvious that there is a mapping between a key value pair like {"placeOfService": "Inpatient"} in the example JSON file, and the column "Place of Service" in the CSV file. But this is not something a computer is likely to figure out for itself. And by no stretch of the imagination is the computer going to guess that it is supposed to be compare the "dateOfService" to see if it is greater than or equal to "Effective Period Start" and less than or equal to "Effective Period End". Somewhere we need to capture the mapping between the columns of the decision table and the keys in the JSON input file.

There a couple of ways of doing this. The first would be to embed this mapping in the code. This is the approach taken in Matteo's Prolog solution where the variables of the JSON input/output terms are unified with the variables of the decision table term, and Pradeep Venkat in his Java solution[3]. However, since I'd explicitly set out with the intention that my solution would not contain any specifics about the decision table, including what the inputs and outputs looked like, I couldn't really start putting column names and keys in my Java code. The obvious solution therefore was to enrich the CSV with the extra information. Indeed, one could take the reasonable view that this information 'should' have in the file to start with. I therefore added in a couple of additional header rows as shown below:

	A	B	C	D	E	F	G	H	I	J	K
1	Place Of Service	Service Type	Plan	Group Size	In Network	Is Covered	Effective Period Start	Effective Period End	Covered in Full	Copay	Coinsurance
2	placeOfService	type	plan	groupSize	inNetwork	isCovered	dateOfService	dateOfService	coveredInFull	copay	coInsurance
3	equals	equals	equals	equals	equals	equals	greaterThan	lessThan	result	result	result
4	Inpatient	dentalAcc	PL123	L	Y	Y	01/01/2015	31/12/2023	N	Copay Mir	N
5	Outpatient	dentalAcc	PL123	L	Y	Y	01/01/2015	31/12/2023	N	Copay Mir	N
6	Office	dentalAcc	PL123	L	Y	Y	01/01/2015	31/12/2023	N	Copay Mir	N
7	Inpatient	dentalAcc	PL123	L	N	Y	01/01/2015	31/12/2023	N	N	Y
8	Outpatient	dentalAcc	PL123	L	N	Y	01/01/2015	31/12/2023	N	N	Y

[3] See https://dmcommunity.files.wordpress.com/2021/02/challenge2021feb.java_.pdf

The second row identifies the key of the value in the JSON input which is to be compared to the value in the column. The third row identifies how the comparison is to be made for the 'input' columns and marks which ones are 'output' ('result').

3.3 Implementing the logic

With these additional header rows, the CSV file now completely describes how to evaluate a case. The next step is to work out how to translate this description into something executable. The simplest approach is to literally translate each row into a rule, and then work through each rule until you find a match or until, until you run out of rules (in which case the output fields are set to 'NOT FOUND', as per the third example in the test case file). This is essentially the approach in Pradeep's Java solution and Matteo's Prolog solution. Pradeep's solution uses linear search, which on such a large table inevitably causes some performance issues. On the other hand, the Prolog unification process is rather clearly efficient.

Having had my fingers burnt by a product which used linear search in the past, I wanted a more efficient way to figure out which (if any) rule applied to a case. Having advised a colleague on managing a 1.7 million row decision table, I had an idea on how to go about it. The answer arguably leaps out when you look at the test case and think 'the first case is about an *Inpatient* service, so I really only want to look at rows where the *Place of Service* is *Inpatient*. So why don't I build a hash table where the key is *Place of Service*, and the value is a list of the rules for which *Place of Service* is *Inpatient*. Now I only have to look through 5501 rows not 16369. But then, the *Service Type* is *adultImmunizations*. So, I can have another hash table just looking at those rows which are already *Inpatient* rules, now I only have 37 to look at. And so on. And then you can see you can do even better than this. Why have several nested hash table? Why not concatenate all the keys and have only one hash table?

Unfortunately, this doesn't quite work. There are a couple of reasons for this. One is to do with blank cells, which we will ignore for the moment and come back to in section 7, the other is we can only use this trick with 'categorical' values, not with 'numeric' ones. For example, while I can be sure that the *plan* in my input must correspond to a value in the *Plan* column, it is highly unlikely that my *dateOfService* will actually coincide with either of the values in the *Effective Period* columns. But by concatenating all the 'categorical' values, instead of a linear search, or six sets of hash table lookups, followed by a linear search on the date rules, I only have one lookup and a linear search on the date rules[4].

The ideas above are encapsulated in a couple of (immutable) classes, **DecisionTable** and **DecisionTableRule**.

The first stage in the use of the tables is to 'compile' the decision table, that is to create the hash table for the categorical attributes, and then the linear rules for numeric attributes and the results values. To this end the constructor of a **DecisionTable** simply takes a list of strings which are the comma separated rows from the CSV file[5]. As

[4] Actually, there is not really a linear search. For the decision table given, only one set of dates exist for each combination of the other input values, so there is only one rule. Were there lots of numeric rules, something a bit more sophisticated than linear search would be worth looking at.

[5] Outside the US and UK, CSV files may use other separators than commas (e.g. semicolons), but my Java code has the separator hardcoded as a comma.

described above the first three rows define the column names for the table, the input attribute names used in the JSON input and the 'type' of the column (*categorical*, *numeric less than or equal to*, *numeric greater than or equal to* and *result*), the remainder are rules. For the specific example table, the values of the first six columns are concatenated to form a key and then an instance of the **DecisionTableRule** class is created to hold the values for the numeric columns and the results columns. The **DecisionTableRule** instance is then stored in a hash table with the key. When a given case is submitted for example:

```
{
      "placeOfService": "Inpatient",
      "type": "adultImmunizations",
      "plan": "PL123",
      "groupSize": "L",
      "inNetwork": "Y",
      "isCovered": "Y",
      "dateOfService": "2019-03-25",
      "coveredInFull": null,
      "copay": null,
      "coInsurance": null
}
```

The **getResults** method of the **DecisionTable** instance is called, this concatenates the categorical attributes of the input to give a key:

```
Inpatient|adultImmunizations|PL123|L|Y|Y|
```

Which is then used to look up the corresponding **DecisionTableRule** instance, this then checks that the date of service is within the appropriate range, and providing it is, it fills in the results columns (*coveredInFull*, *copay* and *coInsurance*).

3.4 Input and Output

Having considered the logic of making the decision, the next issue is to consider how to map the input – which is a JSON file into the format needed by the Decision Table. Since the decision table logic described only deals with one 'case' at a time, some logic is needed to cope with unpacking the test JSON into separate cases. Then for each case one needs to map the case attributes into the attributes used in the decision table. Generally used JSON <-> Java mappings provide this for free, so the JSON is simply parsed into a List of Map objects, and then each Map in the List, is passed to the **getResults** method of the **DecisionTable** instance, which fills in the output columns.

One important simplification on the input/output, is that dates in both the JSON file and the CSV file are in "yyyy-mm-dd" format. This means that rather than having to convert the date values into Date objects, one can compare the dates just using string comparison. Once one moves onto decision tables with numeric values which are not dates, some more code would be needed. Another thing worth noting. In the figure above showing the enriched CSV file, you can see Excel 'helpfully' converts the dates from "yyyy-mm-dd" format, into my local format ("dd/mm/yyyy" for the UK). Which of course causes problems. And things got worse when I looked again at Drools.

4 Solutions in Drools

Having built my custom solution in two and bit days. I decided to go back and have another go with Drools. Now in theory this ought to have been easy. Drools supports decision tables, and I've used them without much trouble in an earlier DMC challenge. However, it took me just as long to get a Drools version working.

After four attempts at downloading different Drools versions and different Eclipse plug-ins, I finally got something which worked[6]. Having overcome these problems, I then had a long tussle trying to get the generate the decision table. Drools does not directly implement decision tables, but rather you create an Excel spreadsheet, which then gets compiled into the Drools internal rule language. The documentation on how to format your decision table leaves a little to be desired, but thankfully, the Eclipse plug in automatically generates an example, from which with a little effort you can figure out what to do. It's basically all about setting up some header columns to define what the attributes are, and how they map to the input object(s), in very much the same manner as the extra rows I added in my custom solution.

However some more issues arose, partly down to Eclipse, partly down to Excel, partly down to the CSV provided and a lot down to crass stupidity on the part of the developer (me). I thought all I would need to do having set up the header columns, would be to open up the CSV file and copy all the rows from this (except the header row), and paste them into the correct place my decision table spreadsheet. The next three hours had me tearing my hair out!

Eclipse caught me out again and again, because if you edit a file in an Eclipse project outside it (ie the Excel spreadsheet), Eclipse doesn't notice. So, when you run your project, it generally uses a cached copy of the spreadsheet and doesn't take account of your edits unless you do a refresh.

Excel as noted above, is generally tempted to be over helpful, messing with things like date formats and converting strings which look like dates into dates (and reformatting them in the process) and strings which look like numbers into numbers. In a game attempt to avoid this problem, the CSV file encloses actual numbers in quote characters (“). Unfortunately doing this screws up the Drools decision table compiler, because the rules see things like “”50%””, which is illegal syntax. Being extraordinarily dense it took me quite a while to discover the errant extra quote marks. However, if I tried simply removing them and copying and pasting, in Excel, Excel insisted on turning 50% into 0.5. Eventually I found that if I removed all the quote characters, replaced all the comma separators by tabs in a text editor, when I did a copy and paste, Excel finally got it right.

And onto the code. Well as I said it should be easy. And once I'd overcome the issues of getting a working installation of Drools and in importing the content of the CSV into the Excel spreadsheet it was. Depending on the implementation – which I'll go into in

[6] Though for the first day, for no obvious reason it continually gave me an alert message that there was an error in the build – even though the code ran fine. Without doing more than restarting my computer a couple of times, the alert now no longer pops up. Some of the issues are probably down to the fact the Drools team are not really maintaining the plug-in. There may be good reason for this. But why not make it clear on the main Drools site? Looking at recent YouTube videos, it seems that the Drools team are actually using VS Code rather than Eclipse for demos, so I'll probably have a look at that sometime as an alternative to Eclipse.

the next section there are literally only three or four lines of Drools code involved. All the other code is about input and output of the JSON test cases.

5 Raw Performance

5.1 Custom Solution

As stated, to start with the idea was to try and set up some kind of yardstick to measure performance against. But what sort of performance should one measure? One is timing how long it takes to make the decision, but where does the making of the decision start and stop. If one is dealing with a web-based service, is it time between the client sending a request to the service, and receiving a reply? If we use this, how much of our answer is down to our ISP, and whoever is hosting the service, as opposed to the actual decision table implementation?

It seemed to me it would probably make sense to start by stripping things back to a bare minimum. So, I built a simple harness which reads the given test case from the file system, parses it into a list of individual cases and then submits each to the decision table engine in turn. But there remain open questions. Do you time from reading the case file from the disk and saving, from parsing the case file from JSON into Java objects, or from the call to the decision table? To illustrate the difference, we can try each.

Due to Java's JIT Optimiser, the performance of the decision engine improves considerably the more it is exercised, so to get a realistic idea of what can be expected, the tests are run with 20,000 executions to start with, against the test JSON containing 10 individual cases. Then timings were taken by executing the engine 1000 times against a test JSON file. To get a bit more of a feel for the marshalling unmarshalling aspects, rather than simply use 10 cases, I build a childishly simply test case generator to create some large test files. The generator simply selects random rows from the CSV files and spits out an JSON file which matches them. This gave me test files of 100 cases, 1000 cases and 10000 cases. To try and iron out scenarios where Windows 10 decided to wander off and spend it's time on other things than my performance test I did five runs of each test.

A typical output was something like the following:

```
Custom
Using warm up case: C:/Projects/DMCommunity/202102/testcases/10.json
Using test case:    C:/Projects/DMCommunity/202102/testcases/1000.json
Warming up count:   20000
Execution count:    1000
Total execution time is: 19.460s
Compile Time 109.903
WarmingUp    0.468
ReadFile     6.280  0.006
ParseCases   3.028  0.003
CoreLogic    0.676  0.001
```

Except the total execution time (which is in seconds), all timing are in milliseconds[7]. These measurements were taken on my rather aging desk top machine which has an i5-4670K CPU @ 3.40GHz processor. As can been seen, the time taken to compile the decision table is only about a tenth of a second, (and this step takes place as the program starts so before the JVM has really warmed up). The average time to process ten cases during the warmup process is about half a millisecond (this is full end-to-end processing, reading in the test file, parsing it, running the decision table, unmarshalling the result and writing to the disk again). In the rows 'ReadFile', 'ParseCases' and 'CoreLogic', the second figure is the time per case, rather than for the whole test file (and since in this case there were 1000 cases per test one thousandth of the first figure). Looking at the last row we see that it takes around two thirds of a millisecond to process 1000 cases once parsed – so somewhat less than a microsecond per case.

The 'compile time' here is the time taken to load the (enriched) CSV table defining the decisions and to build the internal representation. This ought to be constant, but since it only occurs once, when the engine is called for the first time, it is subject to some variability. As can be seen the reading and writing is the most time-consuming process by an order of magnitude or so, and the unmarshalling and marshalling of the JSON about three quarters of the remaining execution time.

5.2 Drools Solution

When trying to perform the same metrics with the Drools solution, there are more options to choose from. Drools provides for Stateful and Stateless sessions, and unlike my bare bone custom solution, the Drools API allows for multiple cases can be submitted to the engine at once. So instead of the three options above, we have up to twelve options for what to measure. The code for Reading and writing the JSON was identical to that used for the custom solution and the JSON code for parsing similar, but instead of converting the JSON to a HashMap, it generates a POJO. So, I decided not to bother measuring the reading and writing and parsing separately for the scenarios where the cases are passed one by one and as a group.[8]

Using the stateful engine a typical result was:

```
Stateful Drools
Using warm up case: C:/Projects/DMCommunity/202102/testcases/10.json
Using test case:    C:/Projects/DMCommunity/202102/testcases/1000.json
Warming up count:   20000
Execution count:    1000
Total execution time is: 89.696s
Compile Time 24926.928
WarmingUp           0.567
ReadFile            18.529 0.019
ParseCases          12.465 0.012
Minimal1AtATime     10.668 0.011
MinimalBulk         11.748 0.012
```

[7] The overall execution time is not important, and potentially misleading. I put it there to give me an idea if I should go and make coffee while waiting for it to run. But in this run the engine processed 3.2 million cases. 200,000 (20,000 x 10) during warm up and then a million (1000 x 1000) with full processing, another million reading the file once and parsing and processing it 1000 times, and third million reading it and parsing it once and processing it 1000 times. All in under 20s, which was a lot faster than I expected!

[8] The stateful engine performance appears better processing one case at a time, the stateless engine processing in bulk, so these are the methods used when gathering the end-to-end logic and the parse + decision table logic for each decision table execution mode.

A corresponding sample output for the stateless engine was:

```
Stateless Drools
Using warm up case: C:/Projects/DMCommunity/202102/testcases/10.json
Using test case:    C:/Projects/DMCommunity/202102/testcases/1000.json
Warming up count:   20000
Execution count:    1000
Total execution time is: 98.505s
Compile Time 23551.990
WarmingUp           0.562
ReadFile            19.425 0.019
ParseCases          13.188 0.013
Minimal1AtATime     20.268 0.020
MinimalBulk         10.811 0.011
```

As before all timings are in milliseconds except the overall execution time. The first thing to notice – and it is extremely noticeable – is that the 'compile time' here is getting on for half a minute. To be fair one can 'pre-compile' the decision table, in which case one would expect the load time for the rules to be considerably quicker[9]. What I found rather more surprising was the difference between using stateful vs stateless execution. Simplistically one would consider the execution of a 'simple' decision table as not involving any state, and hence that a stateless engine would process it faster. However, the statistics indicate otherwise. Using the stateful engine consistently gives better performance when processing cases one at a time. This changes when one does 'bulk' processing where all the cases are submitted to the engine at once, where the two executions are roughly par with the stateless engine seeming to draw ahead a little when there are more cases to simultaneously process.

5.3 Comparison of Drools and Custom Solution

The following tables gather together the raw performance statistics for processing input files containing 10, 100, 1000 and 10000 cases:

		Times in Milliseconds To Process Whole Input File					Times in Microseconds To Process One Case			
		Number of Cases in Input File					Number of Cases in Input File			
Engine	Operation	10	100	1000	10000		10	100	1000	10000
Custom	ReadFile	0.398	0.961	6.153	56.17		39.80	9.61	6.15	5.62
	ParseCases	0.025	0.262	2.992	32.87		2.50	2.62	2.99	3.29
	CoreLogic	0.006	0.065	0.666	9.93		0.60	0.65	0.67	0.99
Stateful Drools	ReadFile	0.443	2.079	18.46	150.13		44.30	20.79	18.46	15.01
	ParseCases	0.045	0.662	12.465	100.85		4.50	6.62	12.47	10.09
	MinimalBulk	0.035	0.543	11.622	101.34		3.50	5.43	11.62	10.13
	Minimal1AtATime	0.032	0.468	10.668	86.13		3.20	4.68	10.67	8.61
Stateless Drools	ReadFile	0.458	1.941	19.055	157.64		45.80	19.41	19.06	15.76
	ParseCases	0.052	0.723	12.942	109.48		5.20	7.23	12.94	10.95
	MinimalBulk	0.038	0.512	10.811	92.63		3.80	5.12	10.81	9.26
	Minimal1AtATime	0.147	1.779	20.268	181.42		14.70	17.79	20.27	18.14

[9] The Drools documentation is typically obscurantist about how exactly to do this. I have managed to do it the past, but I haven't re-figured out how to do it again.

What one can take from these results is that my custom engine is about 5 or 6 times faster at processing individual cases, but that the parsing of the JSON is less efficient. This is probably down to the fact that the custom solution uses a Map to hold the case rather than a POJO, so insertion and extraction is more time consuming. The highlighted figures based on parsing and processing thus look to be the most sensible to use. This also corresponds closely to the response time of any web-based solution ignoring network delays. The custom solution thus is looks to be roughly 2.5 to 3 times more performant than the Drools solution.

6 Trying to Compare Like for Like

6.1 Building and Measuring Performance of a RESTful web service

Building a RESTful service in Java for the first time involves a few hoops, but having done it once, it's easy enough to do again. The usual tactic is to deploy the service on a web server supporting Java Servlets, and then use JAX-RS technology. I used Tomcat-9 for my server and the Jersey JAX-RS libraries. Each service exposes two resources, one which consumes POST requests with JSON in the format of the test case as the body and returns the JSON 'enriched' by having the results fields filled out. The second resource returns a JSON file which embeds the original JSON with the results filled in, but which also includes timing information about how long the request took to process on the server.

An issue which I glossed over in the previous section is loading/compilation of the decision table. If one is processing matters in a 'batch' mode starting up a standalone program and running it to completion, the compilation process is not an important part of the process. If one is running things as a service, it can become so. The service cannot compile the rules for each call. Matteo's Prolog solution avoids this issue because the table is explicitly part of the code and so compiled with the service. However, in both the Custom and the Drools solution compilation takes place as part of the running of the program. To avoid this happening every time, it is needful to build a cache to hold the compiled code between invocations of the service. To this end a simple manager class[10] holds compiled decision tables in a class static, from where they can be fetched when needed.

The custom solution generates a WAR file for the service of ~6Mb, mostly the Jersey jar files, for REST, and the Jackson files for parsing JSON. The custom code is less than 100kb, and that is almost all the CSV file representing the decision table. In contrast the Drools service generates WAR file of ~67Mb, most of which is probably superfluous JAR files, but with little direction from the documentation, it would be very time consuming to work out what is needed and what is not.

As noted above, the nature of the Java JIT optimisation approach, means that the performance of a Java web service improves the more it is used. The challenge points to *Postman* as the recommended way to invoke the deployed web service. However, while *Postman* makes it easy to call/test a service, it doesn't immediately seem to provide for repeated calls. To 'warm up' my web service would I have to keep pressing the send button? No! A Google search pulled up an excellent tutorial on how to do

[10] See Appendix 9.2 – the code for the **DecisionTableMgr** class for Drools is very similar, but a Drools **StatelessKieSession** object is cached rather than a **DecisionTable** object

performance testing with *Postman* by Anna Dolnyk[11]. Using her example as a basis, it turned out to be quite easy not only to make multiple calls, but also to gather and collate the performance statistics too.

As an example of the kind of output one gets, here is the result of a call to a local copy of Tomcat of the RESTful service which uses the Custom solution. The data file has 10 cases, and we are looking at the statistics where Postman has made the call 100 times:

Stage	mean	sdev	median	min	max
Overall	6.190	34.961	3.000	2.000	354.000
Get decision table	0.001	0.000	0.001	0.001	0.002
Get & parse input	3.516	34.610	0.034	0.030	347.878
Process Cases	0.021	0.005	0.019	0.017	0.060
Write Output	0.042	0.029	0.034	0.022	0.228

Here 'Overall' signifies the end-to-end taken for the call, *including* network delays. One interesting aspect of this particular example is that there was very long delay *on the server* on one call. This skewed the results dramatically. In the long call, the *get & parse input* logic took about 10,000 times longer than usual, giving a mean 100 times the median. This suggests that one should pay more attention to the median (or better still the minimum) time than the mean.

6.2 Docker

Having built the RESTful services (one for the Custom solution, one for Drools) I was now in a position compare the Drools and Custom solutions against some of the others using Docker. In particular I was interested to look at Matteo's Prolog solution, and also Jacob Feldman's OpenRules solution[12].

To manage this, I used as a basis for the Custom and Drools solutions a Docker Image supporting Java 11, and Tomcat 9 to act as the REST web service host. The Docker files is very simple:

```
# we are extending everything from a tomcat 9 / java 11 image
FROM tomcat:9.0.44-jdk11-openjdk-slim-buster
MAINTAINER Bob

#ADD the application WAR to tomcat directory
ADD ./dmc202102-custom-web.war /usr/local/tomcat/webapps/

# EXPOSE the port used by tomcat
EXPOSE 8080

# start up tomcat with the image
CMD ["catalina.sh", "run"]
```

And to build and run the image, one simply uses the following Docker commands:

[11] See https://anna-dolnyk.medium.com/performance-testing-with-postman-715fa0d717e3
[12] See https://openrules.wordpress.com/2021/04/12/benchmarking-decision-service/

```
docker build -t dmc-custom .
docker run -p 80:8080 --name dmc-custom  dmc-custom
```

The Tomcat server running in the Docker container, uses port 8080 by default, so we bind this to another port (port 80 in this case) on the host. I built separate images for Drools and my custom solution (they only differ with regard to the WAR file, but they could have been combined into a single one.

Together with the Docker images based on my own solution, I also ran the Docker images provided by Jacob and Matteo, which coincidentally seemed to be the best performing solutions.

Due to a lack of sensible hardware, the Docker Images run in a Virtualised Linux (Ubuntu 20.04.2) machine running on Oracle VirtualBox running on a Windows 10 host. The processor is a i7-6700HQ CPU @ 2.60GHz (so slightly slower than my desktop). As described above the performance tests were carried out using Postman. The requests were sent from my desktop to the Docker containers over my local network. The test JSON containing 10 cases was the original sample provided in the challenge[13], the tests with 100, 1000, and 10000 cases were randomly generated.

Matteo's solution does not provide 'internal' times, so the performance of his solution is 'end-to-end' The following table summarises the results of the comparison of the end-to-end timings:

# cases	Engine	mean	sdev	median	Min	max	Time/Case
10	Prolog	15.160	4.977	13.500	11.000	33.000	1.100
	OpenRules	22.900	7.322	22.000	11.000	62.000	1.100
	Drools	13.790	5.735	12.000	7.000	39.000	0.700
	Custom	16.960	14.317	12.000	7.000	104.000	0.700
100	Prolog	52.120	9.223	53.000	32.000	72.000	0.320
	OpenRules	60.550	14.574	60.000	34.000	158.000	0.340
	Drools	31.970	10.030	29.000	19.000	93.000	0.190
	Custom	29.500	12.412	25.000	17.000	68.000	0.170
1000	Prolog	275.180	10.639	274.000	255.000	311.000	0.255
	OpenRules	320.530	45.508	317.500	268.000	686.000	0.268
	Drools	152.550	71.402	140.000	104.000	817.000	0.104
	Custom	114.970	33.824	102.500	87.000	300.000	0.087
10000	Prolog	2571.100	112.329	2557.000	2447.000	3270.000	0.245
	OpenRules	2776.760	219.517	2735.500	2517.000	4155.000	0.252
	Drools	1051.130	221.594	1015.000	946.000	3166.000	0.095
	Custom	917.200	80.409	893.000	797.000	1181.000	0.080

The final column gives the time per case in the file, based on the *fastest* response of the 100 calls (the value in the min column), rather than the median or average, since this best represents the optimal performance, when the server and the network are not being distracted by other matters.

[13] This is not quite accurate – Jacob's solution uses a slightly different format for the JSON file, so the test files needed to be appropriately modified to accommodate this.

Given I have always thought of Prolog providing elegant, but not necessarily very performant solutions, I was frankly astonished at how well Matteo's solution performs. Though as discussed later, perhaps it is not as surprising as it seems.

As would be expected, the performance improves the more cases packed into a file, as the network latency becomes less of a factor. Processing a million cases in batches of 10,000 takes less than 5 mins, but around 18 mins in batches of 10.

Jacob's solution does provide internal timings of the rule execution so these can be compared with the custom and Drools solutions. These are shown in the following table again the time per case is based on the fastest response:

# cases	Engine	Mean	sdev	median	min	max	Time/Case
10	OpenRules	1.607	1.479	1.210	0.427	10.722	0.043
	Drools	1.969	1.504	1.549	0.586	9.643	0.059
	Custom	0.616	0.654	0.480	0.173	5.391	0.017
100	OpenRules	15.074	10.177	13.444	3.880	96.013	0.039
	Drools	10.524	4.154	9.280	5.088	23.430	0.051
	Custom	3.806	2.831	2.760	1.649	18.220	0.016
1000	OpenRules	98.164	13.662	96.781	69.844	136.999	0.070
	Drools	77.984	19.969	75.405	50.709	213.968	0.051
	Custom	40.133	10.118	38.911	8.386	71.190	0.008
10000	OpenRules	975.153	80.565	973.656	763.912	1216.669	0.076
	Drools	582.062	211.481	548.472	488.120	2598.826	0.049
	Custom	426.451	78.557	403.627	329.617	686.977	0.033

Overall, we can see the performance of the Custom and Drools based solutions is quite a lot slower in the Docker environment than in the 'raw' environment. But the point of the exercise was to execute all the solutions in the same environment to abstract away the variability introduced by the network and operating system. And even with though the underlying hardware is somewhat more sluggish, the performance of all the tools is still very respectable. The Custom solution seems rather more performant than Drools which most the time is a little faster than OpenRules, but the very worst case has a million cases being processed (server side) in only 76s.

6.3 AWS Lambda

As regards a cloud-based deployment, using AWS Lambda seemed the obvious choice, especially as it was something I'd not tried before. After one or two false starts, it was not long before I was able to convert the code I'd used to build a RESTful service for Tomcat into something which I could deploy as a Lambda function. Using the same logic in the Lambda function for generating timings, I was then able to reuse the JavaScript code I'd employed with the Docker tests, to gather and process these. The only change needed to the Postman call was to the target URL. The layers of processing to be gone through are different when using Lambda then when using Docker & Tomcat, but are probably comparable, but since I am contacting a server fifty miles away via a VPN, the network delays will be greater than contacting a machine only a couple of feet away!

Given that the Drools solution is so bulky, I've only deployed the Custom solution up to AWS[14]. The performance figures for this are as below:

# cases	Stats	mean	sdev	median	min	max	Time/Case
10	End-to-end	43.480	7.356	42.000	33.000	67.000	3.300
	Rule Execution	0.229	0.023	0.220	0.202	0.333	0.020
100	End-to-end	54.320	6.961	53.000	44.000	84.000	0.440
	Rule Execution	1.241	2.492	0.792	0.706	19.495	0.007
1000	End-to-end	263.000	26.302	257.000	238.000	389.000	0.238
	Rule Execution	13.368	15.325	5.864	5.342	81.107	0.005
10000	End-to-end	2599.190	129.403	2635.500	2402.000	3035.000	0.240
	Rule Execution	351.451	110.805	441.692	220.768	479.812	0.022

The 'internal' performance figures are broadly similar to what was seen with Docker, suggesting the underlying AWS hardware is broadly equivalent to the one used for those tests. The end-to-end time does not seem unreasonable.

7 Revisiting the Problem and the Custom Solution

The custom solution has some issues. But exactly what they are is a little open to question. In a previous challenge on the simplification of decision tables[15], the principal observation it seemed to me is that trying to implement a decision table without understanding where it comes from, and what it is for, is an exceptionally bad idea.

7.1 What do blank cells in the CSV file actually mean?

This observation extends to the current table. As in the earlier challenge we have a decision table, but we do not have a business context by which we can understand it. In particular, what exactly do the empty cells in the decision table mean? Generally, an empty cell means 'This rule does not depend on this attribute'. But in this case, it is not clear if this is the correct interpretation. If we look in the table, we see the column Group Size is either 'L' or blank. if we look at the example JSON, we see that the *groupSize* attribute is either 'L' or explicitly *null*. This would seem to suggest that 'L' and null are the only values permitted, so it is in effect a binary field. The way one would handle such a field is very different to a 'don't care' rule. Likewise, *isCovered* clearly should be either 'Y' or 'N', but at some time some logic to handle a case where no value has been provided was inserted into the table. Of the three columns where blanks occur only those in the *placeOfService* column seem as if they might really mean 'not relevant', so I felt reasonably justified is assuming that blanks do not mean this, rather that 'null' is a permitted and valid 'value'. But suppose they did, how easily can the functionality of my custom solution be extended to deal with this?

The first step is to identify which columns have blank fields in. There are two approaches on can take to this. One is to demand that columns which have can have blanks are marked as such. So, a categorical field might be 'blank', or 'blank nullable' for example. The second approach would be to determine this dynamically while compiling the decision table. Working out if a column has blanks in dynamically, does

[14] The url is https://a3t1nx8qnj.execute-api.eu-west-2.amazonaws.com/default/SimpleMedicalDecision
[15] See https://dmcommunity.org/2020/10/15/decision-table-simplification/ & https://dmcommunity.org/challenge/challenge-sep-2020/

seem a nicer approach but requires more code changes to the compilation phase. Adding in the logic to make blank cells signify 'don't care' rather than 'test against null' can be very done very quickly (only involving a couple of extra lines of code, and the insertion of an additional literal value), but does lead to a linear search on the values of the attribute which are 'relevant' for other rules, so a more sophisticated approach might need to be sought.

7.2 What do the conflicting rules in CSV file mean?

Apart from the blank cells the meaning of which is unclear, there are explicitly conflicting rules. In Seth Meldon's solution[16] he explores the static analysis capabilities of Corticon. One of the outcomes of the analysis is to identify if a given input might match more than one row in the table. Seth's given example (compare rows 738 to 743 in the supplied CSV with rows 15160 to 15165) *might* be unambiguous because *groupSize* can only be 'L' or null, and the 'null' value in this context is a 'real' value rather than signifying the value is not relevant.

But there are many genuine contradictions in the decision table (e.g. rows 3925 vs 4711 have identical conditions but define different outcomes). It emerged from a DMC Online session[17], that the duplicated rules were intended to manage 'overrides'. From an implementation point of view, to do it this is simply the wrong way to do things if you are using a decision table. But the problem description does not describe or explain how 'overrides' work, nor how the original SQL implementation managed them, so the only reasonable approach is to take a 'first hit' or 'last hit' philosophy in deciding which version of the rule one should use (I have assumed a 'first hit' philosophy).

7.3 Decision Table or Database Table?

A recurring issue which I find in the DMC challenges is whether using a decision table is actually the right way to do things. One of the aspects of my custom solution which I found most pleasing was that (apart from the two additional header rows), is uses the CSV file as the *source of truth*. The CSV file itself is presumably essentially a dump of some database table which formed the heart of the original SQL solution. And why is this so good? Basically, because you can ask questions of an SQL table much more easily than you can ask questions of a decision table. Decision tables are designed to answer only one kind of question. You can ask all kinds of questions of a database table. As a simple example you might want to know how policies P123 and P122 differ. So, under what circumstances do two inputs which only differ in the policy give different outputs? A decision table can't tell you, but a database table can. Keeping the database table (or its proxy the CSV file) as the source of truth and compiling the decision table at the point it is needed, directly from this source of truth, assures the two keep in perfect step.

7.4 Large Decision Tables vs 'Normal' Sized Decision Tables

Many of the solutions submitted to the challenge have a performance which I would regard as really a bit disappointing but being realistic, the tools are probably not well adapted to dealing with very large decision tables. As stated at the start of my submission, the sample decision table is 'simple'. It is characteristic of very big decision

[16] See https://dmcommunity.files.wordpress.com/2021/03/challenge2021feb.corticon.pdf
[17] Visit https://youtu.be/Y4MfyZQ2V7c to see a recording

tables that they are ‘simple’. In the discussion of his solution Jacob alludes to having had to a special algorithm in OpenRules to handle such tables.

Another aspect of the decision table vs database table question is that database designers have spent years figuring out how to best index tables to optimise searches. And for decision tables, the kind of ‘search’ one does is very specific, so there are lots of opportunity for optimisation. The excellent performance of Matteo’s solution may be interpreted in this context. Considering a decision table as a vey simple kind of unification problem plays very much to Prolog’s strengths

7.5 What Problem are we really solving?

In the initial statement of the problem, the test JSON involved submitting a batch of 10 cases to the decision table. In the performance testing section, scenarios where many more were considered up to the point that I was timing how long it took to process a million cases on a serverless cloud-based deployment (four minutes and twenty seconds). But how often are we really calling the service, and with how many cases at a time, one, two, twenty, a hundred? And what is the throughput? Is the service called once a day, once an hour, once a minute, a thousand times a second? Obviously if the number of claims in a batch is small, or the number of calls infrequent, raw performance is much less important than other aspects of way a solution is built.

One guess might be that a batch comprises the individual line items for a single medical bill. In this case the average number of cases in a batch may well be quite small, perhaps only single figures. In a few cases – for example for a prolonged hospital stay - a single medical bill might run to a few thousands of line items, but most will be much smaller. In this is the case the network latency and bandwidth are going to dominate over raw decision table performance. It is also likely that there are likely to be a lot of very ‘common’ bills. For example, ‘diagnosticTesting’ is likely a very common procedure. This observation makes a strategy of using a cache more attractive (as illustrated by Rob Parker’s solution).

And what is the business context? For any given treatment or service, a decision needs to be made about coverage. But the table is clearly incomprehensible as a whole. No one person can sensibly have a holistic view of a 16,000+ rule decision rules. Simple data analysis shows there are only 151 treatments or services, together with 20 policies and 48 possible outcomes. Ending up with so many rules ‘feels’ wrong. There ought at least to be some kind of hierarchical way they can be broken down into smaller more manageable groups of rules. Since it is essentially the policies the business sells, my own idea would be to ask in what way policies differ as they are applied to treatments and services. Having rules focused on the *differences* between policies makes business sense, because it clarifies which one should be sold to a client, it should also make it possible to identify the best way organise and simplify the rules. Harking back to the remarks above, an analysis of this type is greatly simplified using a database like representation.

8 Closing Thoughts

A few closing thoughts on the challenge.

It’s interesting to see how easy/difficult it is to create web-based decision services and/or custom code to manage decision tables. But as a benchmark it’s probably not

the best kind of example. The raw performance of any reasonably efficient decision table engine should be of the order of a few milliseconds at most, arguably (on the basis of the code above) two orders of magnitude less, while network overheads including data marshalling and unmarshalling will generally be several times greater – and far more variable. If your network time is of the order of ~50ms, it scarcely matters if your decision engine takes 2ms or 2µs to execute as regards response times.

For a deployed decision service, it is the end-to-end time that is ultimately important at least from the end user's perspective. So as regards performance, for a web-based service, selection of the hosting environment is probably more important than the selection of the technology used to implement the decision table. The performance of the decision engine itself is really only going to be crucial, if the service has a really high demand, which in the context of the current example, means that the batch size is large (several hundred cases per call). It is only at this point solutions which are highly efficient in terms of CPU cycles, will reduce the overall costs, by reducing the computational requirements on the host, and the execution time begins to impact the overall end-to-end time.

The factors affecting decision service technology choice should really come down to the key factors around any good decision service technology. For example: can I understand how I am making my decisions? can I change how I make decisions quickly? can I monitor the quality of decisions I make? can I reproduce the decision-making process I used on a case which was processed by the version of the decision service in use eighteen months ago?

The various custom solutions to the challenge demonstrate that it's actually pretty easy to build a decision table engine for 'simple' decision tables of the type used here. And it is fair to say that 'most' decision tables one encounters – particularly larger ones (say those with more than about fifty or a hundred rows) are of this simple nature. But do these custom solutions actually meet the 'key factors' above? Or would a 'proper' tool (like IBM ODM, Corticon, Camunda, Open Rules, Blaze Advisor, Drools) meet these needs better. Certainly, both Drools and OpenRules demonstrably have very high performance and offer far more supporting tools and logic than my own very simplistic custom solution.

9 Appendix – Source Code for Custom Solution

The code below is the core logic of the custom solution. The **TableLoader** & **DecisionTableMgr** classes are just concerned with loading and compiling the decision table, it is the **DecisionTable** and **DecisionTableRule** classes are the ones that do the work. Anyone who wants to see the full code – including the test harnesses for running the examples and performance testing, please drop me an e-mail. I may get around to putting it all up on GitHub eventually.

9.1 TableLoader Class

```
package dmcfebchallenge.decisiontables;

import java.io.*;
import java.nio.charset.StandardCharsets;
import java.util.*;

/**
```

```
 * Class to load a decision table stored as a CSV file (specifically with ',' as
 * separator) The current implementation only supports the loading of CSV files
 * within the current classpath/jar file but could in principle be extended to
 * support storage on the web, a file system or a database
 *
 * @author bob
 *
 */
public class TableLoader {
    public ArrayList<String> loadTableRows(String name) {

        ArrayList<String> lines = new ArrayList<String>();
        InputStream inputStream = getClass().getClassLoader()
                .getResourceAsStream(name);

        try (InputStreamReader streamReader = new InputStreamReader(inputStream,
                StandardCharsets.UTF_8);
                BufferedReader reader = new BufferedReader(streamReader)) {

            String line;
            while ((line = reader.readLine()) != null) {
                lines.add(line);
            }

        } catch (Exception e) {
            throw new IllegalArgumentException("file not found! " + name, e);
        }
        return lines;
    }
}
```

9.2 DecisionTableMgr Class

```
package dmcfebchallenge.decisiontables;

import java.util.*;

/**
 * Class to manage decision tables. Idea is that a server may simultaneously
 * support multiple tables. The first time a table is used it will be loaded and
 * compiled, but from then on the compiled version is used. 'Hot' updates are
 * provided by performing a 'reset' on a table causing it to be reloaded when
 * next needed
 *
 * @author bob
 *
 */
public class DecisionTableMgr {
    // HashMap containing all the loaded decision tables, This is a class level
    // static, so loaded decision tables persist as long as the class itself is
    // not unloaded
    private static HashMap<String, DecisionTable> tables =
            new HashMap<String, DecisionTable>();

    /**
     * Get a decision table. If it has not yet been loaded then load and compile
     * it, otherwise return the loaded compiled table</br>
     * Note this is synchronized to prevent concurrent access to the 'tables'
     * map and potential multiple concurrent attempts to compile it
     *
     * @param name table name - this should uniquely identify the table via a
     *             URI or equivalent
     * @return compiled decision table
```

```
     * @throws Exception if the table cannot be loaded or compiled
     */
    public static synchronized DecisionTable getTable(String name)
            throws Exception {
        DecisionTable result = tables.get(name);
        if (result == null) {
            result = loadTable(name);
            tables.put(name, result);
        }
        return result;
    }

    /**
     * Reset a decision table. This clears it from the 'tables' map so it will
     * be reloaded the next time anyone wants to use it. It has no effect if the
     * table is not currently loaded</br>
     * Note this is synchronized to prevent conflicts with the 'getTable'
     * method.
     *
     * @param name table name - this should uniquely identify the table via a
     *             URI or equivalent
     * @throws Exception
     */
    public static synchronized void resetTable(String name) throws Exception {
        DecisionTable result = tables.get(name);
        if (result == null) {
            tables.remove(name);
        }
    }

    /**
     * Load and compile a decision table by name
     *
     * @param name table name - this should uniquely identify the table via a
     *             URI or equivalent
     * @return the compiled table
     * @throws Exception
     */
    private static DecisionTable loadTable(String name) throws Exception {
        TableLoader tableLoader = new TableLoader();
        List<String> tableRows = tableLoader.loadTableRows(name);
        return new DecisionTable(name, tableRows);
    }

}
```

9.3 DecisionTable Class

```
package dmcfebchallenge.decisiontables;

import java.text.*;
import java.util.*;

/**
 * Class to define a simple kind of decision table, which is specifically fire
 * first
 *
 * @author bob
 *
 */
@SuppressWarnings("rawtypes")
public class DecisionTable {
    // these statics are allow checks to ensure that the column types are
    // supported
    // currently don't handle "greaterThanOrEqual" and "lessThanOrEqual"
```

```
private static String[] KNOWN_TYPES = { "result", "equals", "greaterThan",
        "lessThan", "equals N/A" };
private static Set<String> KNOWN_TYPES_LOOKUP;
private static DateFormat dateFormat = new SimpleDateFormat(
        "yyyy-MM-dd HH:mm:ss.SSS");
static {
    KNOWN_TYPES_LOOKUP = new HashSet<String>();
    for (String type : KNOWN_TYPES) {
        KNOWN_TYPES_LOOKUP.add(type);
    }
}

// Private items to identify table
private String identifyTable;

// these private data items hold information about the column names and the
// associated input attributes
private int numCols = 0;
private String[] columnNames;
private String[] inputFields;
private String[] types;
private HashMap<String, String> columnsToTypes = new HashMap<String, String>();
private HashMap<String, String> columnsToInputs = new HashMap<String, String>();

// this structure basically holds the decision table rules
private HashMap<String, ArrayList<DecisionTableRule>> keyedRuleList =
        new HashMap<String, ArrayList<DecisionTableRule>>();

/**
 * constructor which builds the decision table rules using using table rows
 * from CSV definition of decision table</br>
 * row 0 contains the column names of the table</br>
 * row 1 contains the names of the input attributes (in the input JSON)</br>
 * row 2 contains the type of the column (test for equals, greaterThan or
 * lessThan, or a results column)
 *
 * @param tableRows table rows, the first three row define the table
 *                  structure the remainder are the decision rules
 */
public DecisionTable(String name, List<String> tableRows) {
    columnNames = tableRows.get(0).split(",");
    numCols = columnNames.length;
    inputFields = compileSplitRow(tableRows.get(1));
    types = compileSplitRow(tableRows.get(2));
    compileMapColumnTypes();
    int numRows = tableRows.size();
    for (int idx = 3; idx < numRows; idx++) {
        compileParseRule(tableRows.get(idx));
    }
    identifyTable = String.format(
            "\"tableName\": \"%s\", \"createTime\": \"%s\", ", name,
            dateFormat.format(new Date()));
}

public String getIdentifyTable() {
    return identifyTable;
}

/**
 * Evaluate the decision for a given input case.
 *
 * @param caseData hash map taken from the parsing of the input JSON. This
 *                 is updated by the decision table by filling in the
 *                 results attributes
 */
public void getResults(Map<String, Object> caseData) {
    HashMap<String, Comparable> comparison = new HashMap<String, Comparable>();
```

```
    String key = "";
    for (int idx = 0; idx < numCols; idx++) {
        String columnName = columnNames[idx];
        String inputField = inputFields[idx];
        String type = types[idx];
        if (!"result".equals(type)) {
            Comparable value = (Comparable) caseData.get(inputField);
            if ("equals".equals(type)) {
                key += "|" + (value != null ? value : "");
            } else {
                comparison.put(columnName, value);
            }
        }
    }
    getResultsFromRule(key, comparison, caseData);
}

/**
 * override to pretty print a decision table
 */
public String toString() {
    StringBuffer result = new StringBuffer(1000);
    for (String key : keyedRuleList.keySet()) {
        result.append(key + ":\n" + keyedRuleList.get(key));
    }
    return result.toString();
}

/**
 * Rules come in two parts, those where we just do an 'equals' test, and
 * those where test for an upper/lower bound. The equals test is managed by
 * a hash table look up the comparisons by a look up table of comparable
 * values these are delegated to the 'DecisionTableRule' class which also
 * holds the result values if the key and comparable values match
 *
 * @param key              computed key which determines which
 *                         DecisionTableRule list to use
 * @param comparableValues the comparable values for this DecisionTableRule
 * @param resultValues     the results of this DecisionTableRule
 */
private void compileAddRule(String key,
        HashMap<String, Comparable> comparableValues,
        HashMap<String, String> resultValues) {
    ArrayList<DecisionTableRule> ruleList = keyedRuleList.get(key);
    if (ruleList == null) {
        ruleList = new ArrayList<DecisionTableRule>();
        keyedRuleList.put(key, ruleList);
    }
    ruleList.add(new DecisionTableRule(comparableValues, resultValues,
            columnsToTypes));
}

/**
 * Builds mapping tables between table columns names and their linked
 * comparison types and input attributes If the comparison type is not
 * supported, an exception is raised
 */
private void compileMapColumnTypes() {
    for (int idx = 0; idx < numCols; idx++) {
        String columnName = columnNames[idx];
        String inputField = inputFields[idx];
        String type = types[idx];
        if (!KNOWN_TYPES_LOOKUP.contains(type)) {
            throw new IllegalArgumentException(
                    "unknown column type " + type);
        }
        columnsToTypes.put(columnName, type);
```

```
            columnsToInputs.put(columnName, inputField);
        }
    }

    /**
     * Parse a row from the text definition of the table and then call addRule
     * to create the next rule in the decision table
     *
     * @param tableRow string containing the text description of a rule
     */
    private void compileParseRule(String tableRow) {
        String[] columnData = compileSplitRow(tableRow);
        String key = "";
        HashMap<String, Comparable> comparableValues = new HashMap<String, Comparable>();
        HashMap<String, String> resultValues = new HashMap<String, String>();
        for (int idx = 0; idx < columnNames.length; idx++) {
            String columnName = columnNames[idx];
            String type = types[idx];
            String value = columnData[idx];
            switch (type) {
            case "equals":
                key += "|" + value;
                break;
            case "result":
                resultValues.put(columnName, value);
                break;
            default:
                comparableValues.put(columnName, value);
            }
        }
        compileAddRule(key, comparableValues, resultValues);
    }

    /**
     * Helper which splits a table row checking it is the correct size
     *
     * @param row row to split
     * @return string array with column values
     */
    private String[] compileSplitRow(String row) {
        String[] result = row.split(",");
        if (numCols != result.length) {
            throw new IllegalArgumentException("found " + result.length
                    + "columns instead of  " + numCols + " in " + row);
        }
        return result;
    }

    /**
     * Having computed the key defined by the 'equals' fields, we look up the
     * possible comparisons and determine the results. If no rule matches this
     * is handled by the undefinedResult method
     *
     * @param key        lookup key for comparison fields
     * @param comparison comparison values from input case
     * @param caseData   input case to be updated with the results
     */
    private void getResultsFromRule(String key,
            Map<String, Comparable> comparison, Map<String, Object> caseData) {
        ArrayList<DecisionTableRule> ruleList = keyedRuleList.get(key);
        if (ruleList != null) {
            for (DecisionTableRule rule : ruleList) {
                Map<String, String> results = rule
                        .getMatchingResults(comparison);
                if (results != null) {
                    setRuleResult(caseData, results);
                    return;
```

```
                }
            }
        }
        setUndefinedResult(caseData);
    }

    /**
     * Update the caseData by setting the values of the results columns to the
     * values determined by the rules
     *
     * @param caseData input case
     * @param results  determined results
     */
    private void setRuleResult(Map<String, Object> caseData,
            Map<String, String> results) {
        for (int idx = 0; idx < numCols; idx++) {
            String columnName = columnNames[idx];
            String inputField = inputFields[idx];
            String type = types[idx];
            if ("result".equals(type)) {
                caseData.put(inputField, results.get(columnName));
            }
        }
    }

    /**
     * If no rule fires, the results fields of the case are all set to NOT FOUND
     *
     * @param caseData input case
     */
    private void setUndefinedResult(Map<String, Object> caseData) {
        for (int idx = 0; idx < numCols; idx++) {
            String inputField = inputFields[idx];
            String type = types[idx];
            if ("result".equals(type)) {
                caseData.put(inputField, "NOT FOUND");
            }
        }
    }

}
```

9.4 DecisionTableRule Class

```
package dmcfebchallenge.decisiontables;

import java.util.*;

/**
 * Class to define a rule in a simple decision table
 *
 * @author bob
 *
 */
@SuppressWarnings("rawtypes")
public class DecisionTableRule {
    // local copy of the mapping from the name of a column to the type of
    // comparison
    // it uses
    private HashMap<String, String> columnsToTypes;

    // these map the comparison and results values to their corresponding values
    // for
    // this row/rule of the decision table
    private HashMap<String, Comparable> comparableValues;
    private HashMap<String, String> resultValues;
```

```
    /**
     * Constructor simply squirrels away the values for comparisons and results
     * with the column type mapping
     *
     * @param comparableValues
     * @param resultValues
     * @param columnsToTypes
     */
    public DecisionTableRule(HashMap<String, Comparable> comparableValues,
            HashMap<String, String> resultValues,
            HashMap<String, String> columnsToTypes) {
        super();
        this.comparableValues = comparableValues;
        this.resultValues = resultValues;
        this.columnsToTypes = columnsToTypes;
    }

    /**
     * This compares the input values to the rule values for each comparison
     * column to see if they match this 'rule'. If they do, they return the
     * results
     *
     * @param inputs
     * @return the result values map for a match, null for a mismatch
     */
    public HashMap<String, String> getMatchingResults(
            Map<String, Comparable> inputs) {
        for (String key : comparableValues.keySet()) {
            Comparable columnValue = comparableValues.get(key); // value to test
                                                                // against
            if (columnValue != null) { // null means value is unimportant
                Comparable inputValue = inputs.get(key); // value to test
                                                         // against
                String type = columnsToTypes.get(key); // test type
                @SuppressWarnings("unchecked")
                int comp = columnValue.compareTo(inputValue);
                if (type.equals("greaterThan") && comp > 0) {
                    return null;
                } else if (type.equals("lessThan") && comp < 0) {
                    return null;
                } else if (type.equals("equals N/A") && comp != 0) {
                    return null;
                }
            }
        }
        return resultValues;
    }

    /**
     * override to pretty print a decision table rule
     */
    public String toString() {
        StringBuffer result = new StringBuffer(1000);
        for (String key : comparableValues.keySet()) {
            result.append(String.format(
                    "Field: %s - Comparator: %s - Value: %s\n", key,
                    columnsToTypes.get(key), comparableValues.get(key)));
        }
        for (String key : resultValues.keySet()) {
            result.append(String.format("Result Field: %s - Value: %s\n", key,
                    resultValues.get(key)));
        }
        return result.toString();
    }
}
```