# Data Cleaning and Validation in Ruby

Version: January 2016

# Data Cleaning and Validation in Ruby

This document briefly outlines some techniques and approaches to data cleaning and data validation using the Ruby survey data processing system.

# INTRODUCTION

Ruby can handle all processing of survey case data. Even though this is optional – you could do all case data processing in Quantum, Merlin, UNCLE, CfMC, even SPSS and more recently IBM Data Collection, etc, and employ Ruby simply as a desktop analysis tool – it is nonetheless recommended that all such processing is done within Ruby. The advantages of this are

- All processing is transparent to the analyst
- Far easier for DP to diagnose data-related issues
- Avoids daisy-chaining applications
- Post-field, all participants from DP through to researchers, account managers and end-clients, all use Ruby (although in different ways) on the same case data

A quick description of Ruby's scope would be Quantum + Quanvert, or IBM Data Collection processing + Survey Reporter. To this we also add E-Tabs/Rosetta Stone-type functionality for MS Office deliverables, fuzzy-logic auto-coding of verbatims, and multi-variant analyses such as perceptual maps, CHAID, cluster analysis and regression.

Data processing in Ruby is implemented using scripting or programming languages such as VBScript, JScript, VB.Net, CSharp, etc. You can even use the macro facility in Excel – in short, any modern programming environment will do. If using VBScript, then programming for Ruby looks very much like programming for IBM Data Collection, or for Survey Reporter tables, except that the Ruby scripts are much shorter and a great deal easier to understand.

Most of our clients use either VBS or VB.Net. Note that generic and free programming environments can be used, unlike IBM Data Collection.

## Data Cleaning by Direct Edits

By data cleaning edits, I mean over-writing case data either conditionally or arbitrarily *in situ*. Another variable to carry the clean data is not required. The parameters to the Clean command are

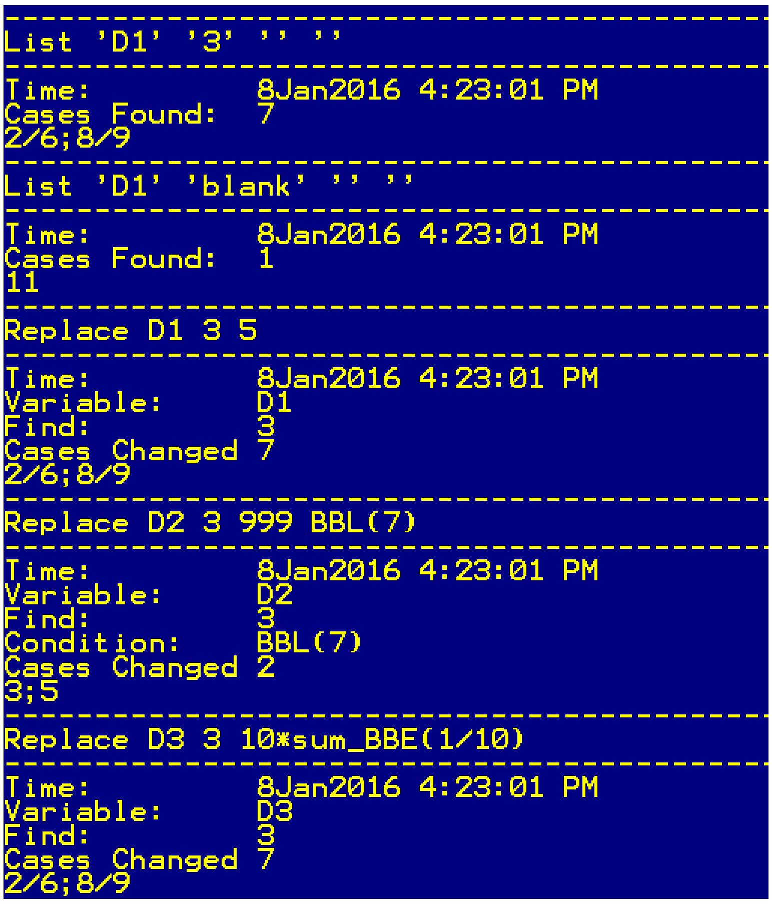
<target variable>, <find scope>, <replacement value>, <condition>, <log file name>

Some example cleaning statements are:

```
'' List all cases with a code 3
        rub.Clean "List",    "D1",  3

'' List all blank cases
        rub.Clean "List",    "D1",  "blank"

'' Replace 3 with 5
        rub.Clean "Replace", "D1",  3,      5

'' Replace 3 with 999 if BBL is 7 (true cases 3,5,11)
        rub.Clean "Replace", "D2",  3,      999,             "BBL(7)"

'' Replace 3 with 10*sum of codes 1/10 in BBE
        rub.Clean "Replace", "D3",  3,      "10*sum_BBE(1/10)"

'' Replace 3 with sum of all codes in BBE / 10 if BBL is 7
        rub.Clean "Replace", "D4",  3,      "sum_BBE(*)/10", "BBL(7)"

'' Replace blank line with sum of all codes in BBE / 10
        rub.Clean "Replace", "D5",  "blank", "sum_BBE(*)/10", "BBL(7)"

'' Replace codes 1/3;5 with 99
        rub.Clean "Replace", "D6",  "1/3;5", 99

'' Replace everything matching target line structure if D7 is anything but empty
        rub.Clean "Replace", "D7",  "any",   "sum_BBE(*)/10"

'' Replace everything including blank lines
        rub.Clean "Replace", "D8",  "all",   "sum_BBE(*)/10"

'' Replace 3 with explicit missing code
        rub.Clean "Replace", "D9",  "3",     "missing"

'' Replace 1 with 101, 2 with 102, 3 with 103
        rub.Clean "Replace", "D10", "1/3",   "101/103"

'' Remove 1 and 3
        rub.Clean "Remove",  "D11", "1;3"

'' Replace all data items with missing code
        rub.Clean "Replace", "D12", "any",   "missing"

'' Replace all data items with missing code if BBL is 7
        rub.Clean "Replace", "D13", "any",   "missing",       "BBL(7)"

'' Replace everything including empty cases with explicit missing code if BBL is 7
        rub.Clean "Replace", "D14", "all",   "missing",       "BBL(7)"

'' Emit 999 at all blank lines
        rub.Clean "Emit",    "D15", "blank", 999)

'' Emit 999 if BBL is a 7
```

```
        rub.Clean "Emit",    "D16", "",      999,             "BBL(7)"

'' Emit the sum of BBE codes / 10 if BBL is a 7
        rub.Clean "Emit",    "D17", "",      "sum_BBE(*)/10", "BBL(7)"
```

Each CleanVar call writes the edited cases to a log file.  There can be many different logs, and log files can be specified per clean as the last parameter. The log output for the above first several examples is



```
----------------------------------------
List 'D1' '3' '' ''
----------------------------------------
Time:          8Jan2016 4:23:01 PM
Cases Found:   7
2/6;8/9
----------------------------------------
List 'D1' 'blank' '' ''
----------------------------------------
Time:          8Jan2016 4:23:01 PM
Cases Found:   1
11
----------------------------------------
Replace D1 3 5
----------------------------------------
Time:          8Jan2016 4:23:01 PM
Variable:      D1
Find:          3
Cases Changed  7
2/6;8/9
----------------------------------------
Replace D2 3 999 BBL(7)
----------------------------------------
Time:          8Jan2016 4:23:01 PM
Variable:      D2
Find:          3
Condition:     BBL(7)
Cases Changed  2
3;5
----------------------------------------
Replace D3 3 10*sum_BBE(1/10)
----------------------------------------
Time:          8Jan2016 4:23:01 PM
Variable:      D3
Find:          3
Cases Changed  7
2/6;8/9
```

## DATA CLEANING BY CONSTRUCTING

Constructing creates a new variable from existing variables. This is appropriate when researchers need access to the original case data, exactly as collected. There are a great many ways to construct new variables in Ruby. For cleaning-type operations, such as unreversing mis-assigned codes, a DefCon (Define a Construction) call is usually employed. For example

```
DefCon "cQ1", "Brand Awareness"
      AddItem 1, "Q1(1)",   "Brand 1"
      AddItem 2, "Q1(3)",   "Brand 2"
      AddItem 3, "Q1(2)",   "Brand 3"
ConClose
```

This creates a new variable, cQ1, which has a code 2 where the original Q1 had a 3, and a 3 where the original Q1 had a 2.

If this clean was required only for a fixed period of time, say wave 1, then the filters would be

```
DefCon "cQ1", "Brand Awareness"
      AddItem 1, "Q1(1)",                             "Brand 1"
      AddItem 2, "Q1(3)&Wave(1)|Q1(2)&~Wave(1)",    "Brand 2"
      AddItem 3, "Q1(2)&Wave(1)|Q1(3)&~Wave(1)",    "Brand 3"
ConClose
```

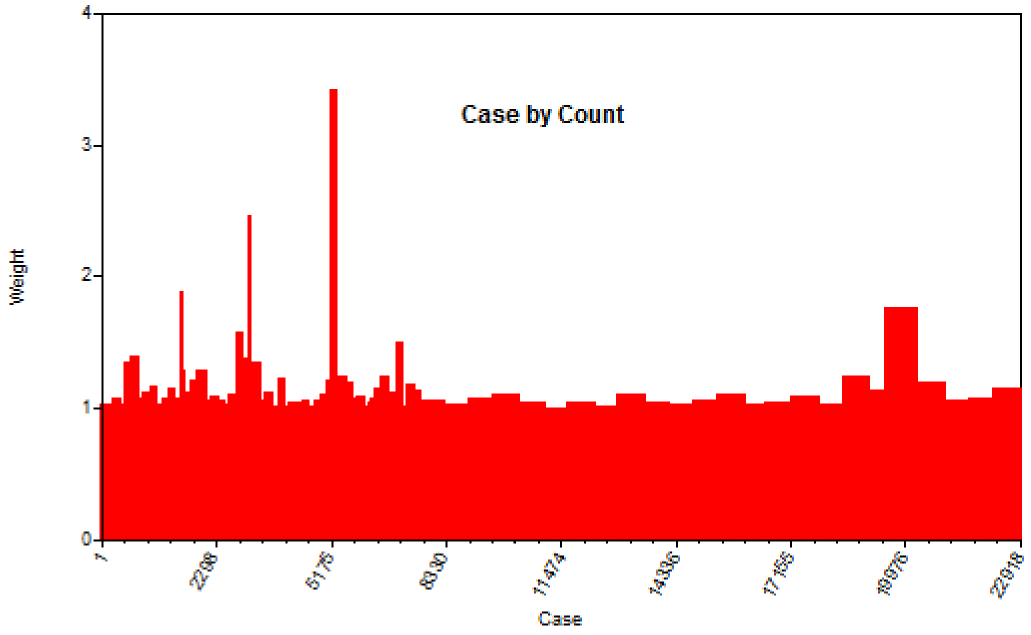
& = Boolean AND, | = Boolean OR, and ~ = Boolean NOT, as is common in many programming languages. There is no practical limit to the complexity of the filter expressions.

*DefCon*, *AddItem* and *ConClose* are part of the Ruby DP language, implemented in a VB.Net library. In effect (and a few have done this), if you do not like our DP language, then you can always design your own from the Ruby APIs in the programming language of your own choice. The Ruby syntax is the same in all programming environments – eg the filter expression "Q1(3)&Wave(1)|Q1(2)&~Wave(1) ". Learning Ruby DP is more a matter of understanding the quoted syntax (which is sent on to Ruby for evaluation or action) than of learning a computer language.

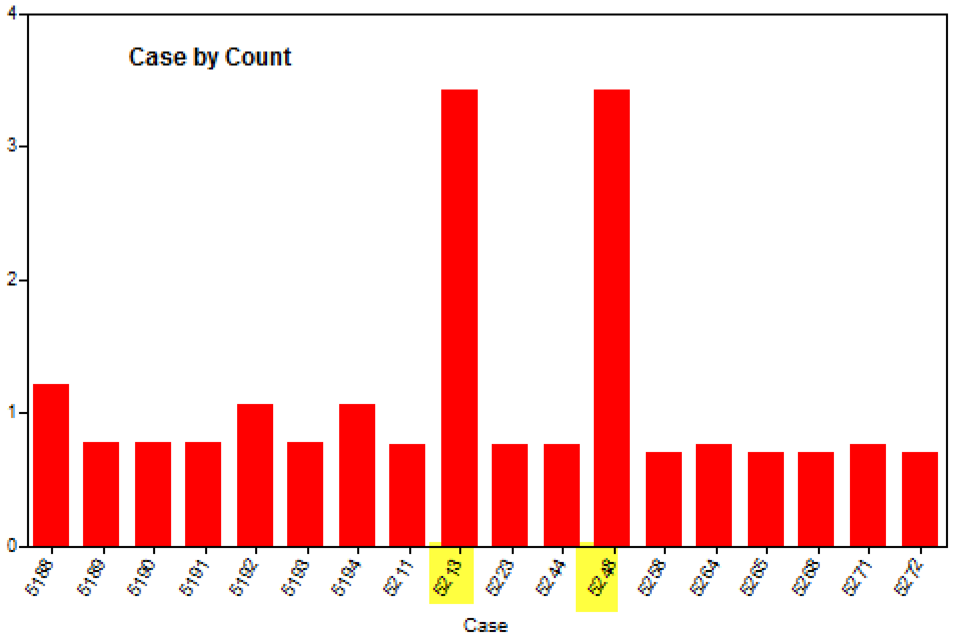
## DATA VALIDATION

Data validation confirms that the case data makes sense, and that there is data where it is expected, and that there is no data where it is not expected.  Here are a few of the techniques commonly employed (there are many more).

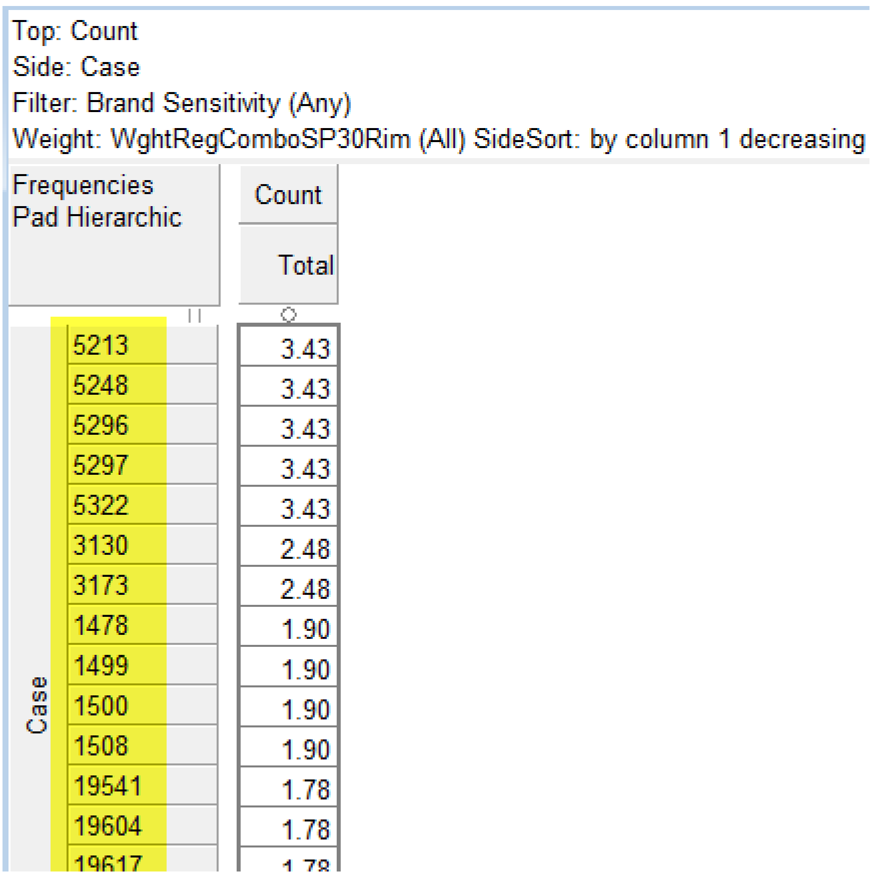
### Histogram of all weight values



The job has 22,918 cases, and the weight value for each case is plotted on the Y axis.  The weights are unstable up to about case# 7,000, and around case# 20,000. To identify the high weights, either zoom in by dragging from either end of the X axis:



(shows that case IDs 5213 and 5248 have excessive weights), or by converting to a table, and sorting in reverse order:



Top: Count
Side: Case
Filter: Brand Sensitivity (Any)
Weight: WghtRegComboSP30Rim (All) SideSort: by column 1 decreasing

| Frequencies Pad Hierarchic | | Count Total |
|---|---|---|
| Case | 5213 | 3.43 |
| | 5248 | 3.43 |
| | 5296 | 3.43 |
| | 5297 | 3.43 |
| | 5322 | 3.43 |
| | 3130 | 2.48 |
| | 3173 | 2.48 |
| | 1478 | 1.90 |
| | 1499 | 1.90 |
| | 1500 | 1.90 |
| | 1508 | 1.90 |
| | 19541 | 1.78 |
| | 19604 | 1.78 |
| | 19617 | 1.78 |

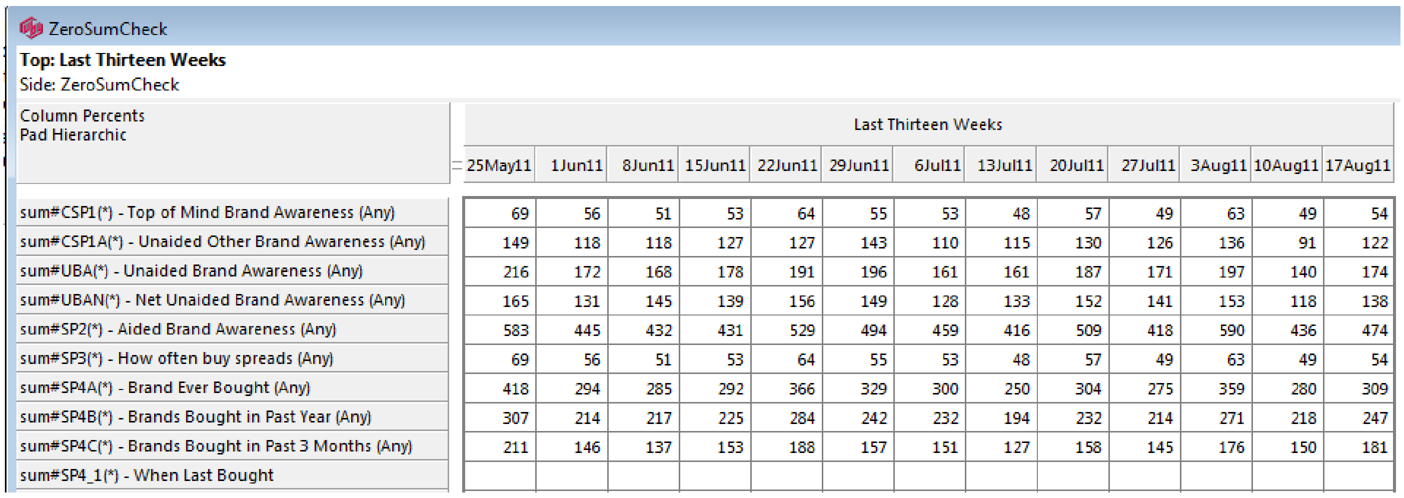
To the extent that most other cross tabulators would not allow a table with 22,918 rows, this is not a common technique.

## ZeroSumCheck Table

This is a table of any banner, usually something like the last 13 weeks, by every variable in the job, where only the marginals are displayed. For categorical variables, the count of codes is shown, and for quantitative or uncoded variables, the sum of values. The row labels show both the syntax form and the plain English form, so that correctness in the variable labelling can be confirmed.

The intention of this table is to determine where variables sum to zero. It frequently catches missing verbatims because the human-coded data had not been remerged with the main case data, and is generally useful for spotting any unexpected changes in the response patterns, and for confirming that new variables are collecting when they ought to, and that retired variables are not.

Some example output is



ZeroSumCheck

**Top: Last Thirteen Weeks**
Side: ZeroSumCheck

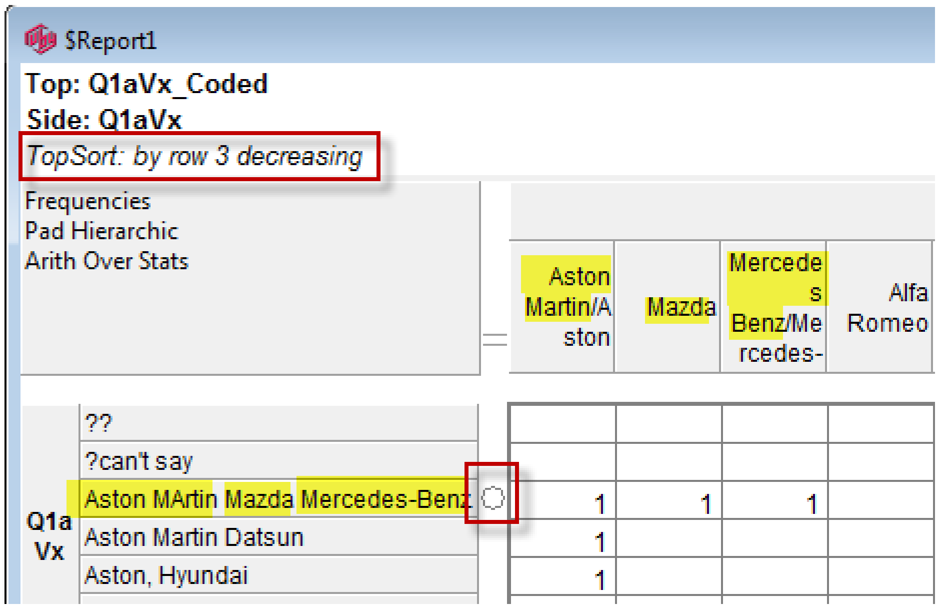
Column Percents
Pad Hierarchic

| | Last Thirteen Weeks | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 25May11 | 1Jun11 | 8Jun11 | 15Jun11 | 22Jun11 | 29Jun11 | 6Jul11 | 13Jul11 | 20Jul11 | 27Jul11 | 3Aug11 | 10Aug11 | 17Aug11 |
| sum#CSP1(*) - Top of Mind Brand Awareness (Any) | 69 | 56 | 51 | 53 | 64 | 55 | 53 | 48 | 57 | 49 | 63 | 49 | 54 |
| sum#CSP1A(*) - Unaided Other Brand Awareness (Any) | 149 | 118 | 118 | 127 | 127 | 143 | 110 | 115 | 130 | 126 | 136 | 91 | 122 |
| sum#UBA(*) - Unaided Brand Awareness (Any) | 216 | 172 | 168 | 178 | 191 | 196 | 161 | 161 | 187 | 171 | 197 | 140 | 174 |
| sum#UBAN(*) - Net Unaided Brand Awareness (Any) | 165 | 131 | 145 | 139 | 156 | 149 | 128 | 133 | 152 | 141 | 153 | 118 | 138 |
| sum#SP2(*) - Aided Brand Awareness (Any) | 583 | 445 | 432 | 431 | 529 | 494 | 459 | 416 | 509 | 418 | 590 | 436 | 474 |
| sum#SP3(*) - How often buy spreads (Any) | 69 | 56 | 51 | 53 | 64 | 55 | 53 | 48 | 57 | 49 | 63 | 49 | 54 |
| sum#SP4A(*) - Brand Ever Bought (Any) | 418 | 294 | 285 | 292 | 366 | 329 | 300 | 250 | 304 | 275 | 359 | 280 | 309 |
| sum#SP4B(*) - Brands Bought in Past Year (Any) | 307 | 214 | 217 | 225 | 284 | 242 | 232 | 194 | 232 | 214 | 271 | 218 | 247 |
| sum#SP4C(*) - Brands Bought in Past 3 Months (Any) | 211 | 146 | 137 | 153 | 188 | 157 | 151 | 127 | 158 | 145 | 176 | 150 | 181 |
| sum#SP4_1(*) - When Last Bought | | | | | | | | | | | | | |

The first row tells me that the variable CSP1 is labelled as *Top of Mind Brand Awareness*, and that the number of responses in each week was between high 40s and high 60s. If the value for 17Aug11 was 400, or 7, then that would most likely indicate a problem, because those values are completely out of character with the past behaviour. Similarly, high code counts would be expected for *Aided Brand Awareness*, so a very low count anywhere would be a surprise. The final blank row tells me that this variable has been retired from the job, and if I saw any counts at all for *When Last Bought*, then that would indicate an error somewhere.

## Confirm Verbatim Coding

Verbatims are a frequent source of problems, especially in tracking jobs where different coders may all work on the same job at different times. In most other systems, the raw verbatims are not disclosed as crosstabbable variables, because the number of unique responses can get very high. As per the Weight histogram chart above, Ruby can do tables with tens of thousands of rows (or columns), so tables of raw verbatim by coded verbatim are trivial to specify, and given such a table, analysts can immediately validate for the levels of accuracy. For example, this table has the raw verbatims down the side, and the coded categories across the top, with the columns sorted to the left on the third row:



$Report1

**Top: Q1aVx_Coded**
**Side: Q1aVx**
*TopSort: by row 3 decreasing*

Frequencies
Pad Hierarchic
Arith Over Stats

| | | Aston Martin/Aston | Mazda | Mercedes Benz/Mercedes- | Alfa Romeo |
|---|---|---|---|---|---|
| Q1a Vx | ?? | | | | |
| | ?can't say | | | | |
| | Aston MArtin Mazda Mercedes-Benz | 1 | 1 | 1 | |
| | Aston Martin Datsun | 1 | | | |
| | Aston, Hyundai | 1 | | | |

This tells me that the raw response text *Aston MArtin Mazda Mercedes-Benz* has been correctly coded for all three brand mentions.

It is standard practice in Ruby jobs to allow all raw verbatims to be present, and they can be used in the same ways as any categorical.

## Tracking Updates

For continuous tracking jobs, there is no limit to the number of things which can go wrong at an update, so we have exhaustive procedures in place to catch processing errors before any damage can be done.

1. The current import file (eg *.sav) is compared to the last import file to ensure that the contents are compatible, and that any new/dropped variables are listed so that the operator can see if the field supplier has made unscheduled or arbitrary changes
2. The Ruby data storage is then checked for case integrity to ensure that there has been no inadvertent damage since the last update (user-error or machine error)
3. At the import itself, the post-import Ruby job is automatically compared to the pre-import job and every difference in code frames or the variable set is reported via an audit file
4. Post-import tests include:
   a) ZeroSumCheck table (as above)
   b) Tables of collection period by weights to ensure that the weights have been correctly calculated within each data-collection period
   c) Statistical tests on the weight variable(s) for standard error, standard deviation, and mean
   d) Detailed reports on the weight calculations, similar to the reports created by Quantum
   e) Histograms of the actual weight values (as above)
   f) Histogram of the wave ID to ensure that the case data is all in chronological order
   g) Any other tables as appropriate to ensure that various check-sum relationships pertain, eg that Top of Mind + Unaided Other = Total Awareness – this ensures that the human coders have not multiply-coded the same brand, or that the same brand is in both Top of Mind AND Unaided Other

For complex tracking operations, all the above steps are scripted.  This ensures that no step can be accidentally omitted.

[end of document]