Paper 118-26

Catch the Errors Early - On Generating Checking Troutines Automagically

Markku Suni, Sampo Plc, Turku, Finland

ABSTRACT

It is widely accepted that a company needs a Data Warehouse. Equally widely accepted is the principle that a company should not warehouse dirty data. Only clean and error-free. To define "error-free" can be quite a demanding task. Even more demanding task is maintaining all sorts of checking routines that find the errors in the input. A method was developed to generate checking routines automagically. This paper discusses our methods, means, aims and experiences on generating checking routines automagically.

INTRODUCTION

Reporting - the art of providing decision makers with the information they need - is a function very central to data processing. During the years people have been using many names about and around reporting, and the development of tools has been immmense. First we spoke about listings, then reports. We had Report Program Generator (RPG) and Report Generator in COBOL. Later on we spoke about more total concepts like MIS, EIS, and many others. Lately we have been building huge Data Warehouses, the main purpose of which is to get information OUT to the reports to support decision-making. In most of these concepts - including the most current, Balanced Scorecard - reporting is the central issue. Always has, always will.

Reporting is the key, and reporting correctly and reliably is obviously the main requirement. The operational (or production) databases contain a lot of information, but as we all so painfully well know, they are not well suited for reporting. The databases have not been designed for reporting, which is why they make life so difficult for someone dealing with reporting. Enter the recent concept of Data Warehouse, something which is designed and built for reporting. After a Data Warehouse has been built, the reporting problems are not over, but a company has a basis for its reporting - provided - and this can be a problem - the data in the Data Warehouse is clean and correct.

Production databases always contain errors. These errors do not harm production runs, but they can seriously affect reporting. For instance, assume the price of a car should be 10.000, but it is stored as 10.000.000 instead. Now, WHAT is the value of our storeroom?

To make sure that its Data Warehouse contains only clean and reliable data, Sampo did put a lot of work in creating a reliable, yet simple mechanism for catching all possible errors in the production data that was going in the Data Warehouse. The incoming data is carefully checked by a program (eventually dozens of programs, as there are dozens of possible sources for the data). It is clear that these checking programs are good and useful only if they can be kept up-to-date. Therefore a semi-automatic or fully automagic system was called for. For each data source there is a number of rules that the correct data should satisfy. For instance, a rule could state that for this kind of insurance policy, for this geographical area and beginning during this time period, the payment should be within these limits. Another rule could state that the change between last year and this year should be within this percentage. For each input material there are many rules, defined by those who know the material well and updated every so often. After each change of rule the maintainer presses a button (clicks a point with the mouse) within the rule maintaining application, and a routine automagically generates a checking program for this material, then saves this program in the library for production programs. Thus the checking programs are always up-to-date even though **no programmer is needed to maintain them.**

This paper discusses the mechanics of generating the checking programs, the advantages of the method and the lessons learned.

WHO NEEDS ERRORS

What is an error? In order to discuss errors we should first define what is meant by an error. In our everyday language we use the word *"error"* to refer to erroneous information in or outside computers, wrong or bad decisions, software or hardware malfunction etc. Even if we limit ourselves to the information stored in a database, we find that "error" is not a word exactly defined. In a customer file, for instance,

- the customer's name may have been misheard and thus typed in other way that it should be,
- the customer may have forgotten the make of his or her car and given the make of the previous car
- the customer may semi-deliberately have told the value of his or her house greater than it really is
- there was a slip of finger when typing which caused the price of the insurance policy to be 1999 instead of 1000
- the value of the customer's house has become higher or lower, but the change is not reflected in the database
- a bad spot on a diskette (or line noise during file transfer) caused a piece of information to be left empty
- an erroneous program updated some insurance policy information differently in certain cases
- until we get a certain piece of information, we put in a default value, which is still in place
- and so on, ad infinitum.

In all these cases, the information in the database does not conform to the real life situation, even though the reason for

the situation varies. In some of these and other cases we should really use some other expression, but generally we speak about erroneous information regardless of what is the cause of it. In the context of this paper we use the short word "error" meaning erroneous information in a database.

Nobody likes errors, nobody wants to have them, yet we are always burdened with errors. It might be a wise plan to try to prevent errors from creeping into databases. Unfortunately, looking at the list above we see immediately that it is impossible to guarantee that our databases are error-free. We can add all sorts of error-checks in the code of the application used to input the data, and this is naturally done to an extent possible or reasonable.

In a typical transaction processing system the production must not be stopped, not even slowed down. As the system contains millions and millions of records, most of which just stay intact on the disks, erroneous information is not so bad a problem as it might sound. Let us say, by slips of busy fingers my name has been somewhat mistyped, I have been marked down as a woman, and the value of my house has been stored as hundred times what it should be. That causes no problem in the transaction processing system. Everything runs normally, including the billing application, which sends me a bill. When I receive the large bill, I react, phone the company, and tell them what I think about them. I also give them the correct information and make sure they get it right. Error is fixed, I get a new bill, and life goes on. No problem. Meanwhile, back at the farm, a manager wants to get a report about the insurance policies sold lately and about the value of the property insured in certain area, where the company has insured some 50 houses. Because of my too valuable house the manager gets a very misleading report, draws most unfortunate conclusions about the situation and makes most unfortunate decisions about sales efforts in this geographic area.

The conclusions are obvious:

- even though it would be most desirable to run an error-free production system, it is not possible
- even though the production systems do contain errors, the reports used in decision-making should be error-free to the extent possible, preferably totally.

This means that if the reports are created from the Data Warehouse, the information put into it must be correct. Said and done. The only thing missing is the doing.

CHECKING OR CORRECTING OR BOTH?

The goal being to guarantee that the Data Warehouse (DW from now on) only contains correct information, the question is: how do we guarantee it. What should we do? Obviously there will be a program for each data source. This program will read the data, possibly do some sorts of conversions and then write the information into the DW. If there is anything to be done for the errors, it should be done together with this input program. There are basically three reasonable courses of action:

Check the information and

- correct it on the fly so as to write only correct information to the DW, or
- give a thorough error report and in case there are errors, stop the process, or
- give a thorough error report and write the information in the DW, but make a note of the errors

It is obvious that the first plan is not realistic. Even if it were possible, correcting the errors at this point would leave the erroneous information in the production databases, which is not good idea. Instead, we should point out the errors to the people maintaining the production databases to give them a chance to correct the original data. Then the next input program will get correct information.

Should we allow errors in the DW or not? If not, this may lead to a prolonged time entering information into the DW thus delaying reporting - perhaps unnecessarily. It seems reasonable to let erroneous information in the DW so that a user can create reports based on the material, if there is an urgent need to do so. Yet there should be an indication that this information is from a given date and that it contains errors, which leaves it to the user to decide whether to wait for the correct information or use what is available. We followed this course.

In case checking programs find errors, they should report the errors as clearly as possible. Then the error messages are sent to the persons responsible for the original material, because it is their job to correct the errors thus found. There is a possibility that due to a change in environment, these errors are no longer errors, and the checking program should be changed to conform to the new situation. Reasons for this might be inflation, deflation, new currency rates, new insurance policies, etc.

THE GENERAL IDEA

Since there are dozens of twisty input materials, all different, there would be a need to write and maintain dozens of checking programs. In the long run the documenting and maintaining of them will become an unnecessary burden, which would tie up too many programmers needed elsewhere. There is a need to generate and maintain the checking programs automagically. For this a new application was written. The main idea is rather simple:

- Create a set of relational tables to contain rules the correct input material must satisfy
- write a simple-to-use application to maintain those rules by someone who knows the business
- within this application write a generator program to generate a checking program based on the rules

- after the person maintaining the rule base has updated some rule, he or she just presses (clicks) a button thus causing the corresponding checking program to be generated and stored into the checking program library
- when the material in question is read in the next time, there will be a new up-to-date checking program.

All this calls for careful design and planning. If it could be done, it could save a lot of work and make it possible to have as error-free DW as possible within reasonable costs.

THE RULE BASE

The design of the rule base could lead to several different implementations. One could for instance think about writing the rules in the form of program code. This form would be handy to a programmer, but obviously not for an insurance expert. It was felt better to have the rules in a form more familiar to the insurance specialists.

In the end this kind of a rule base can become rather complicated, but the beginning is simple. In our case the incoming material consists of all sorts of insurance data. There are classification, or class variables, such as the coverage type, geographical sales area, customer segment, etc. Besides these there are analysis variables, such as price, amount of indemnity. There are variables that could be considered as analysis or as class variables, such as the date of the beginning or end of the term, and so on.

All in all, we need the following for each rule:

- the name of this rule for later reference - **rule id**
- the name of the material this rule is used to check
- possible error message text
- name(s) of the class variable(s) - there may be one or several class variables per rule
- name of the analysis variable
- values to be checked
- type of the rule

These are the main ingredients of each rule. Besides these there is some other information to help in maintaining the rule base and to ease the generating. There are different types of rules, for instance

- the analysis variable must be larger than a given limit
- the analysis variable must be between given limits
- the analysis variable must have a value belonging to a given set of values
- the analysis variable must not differ from another variable (like last year's value) more than a given percentage

For each of the rules there may be varying number of class variables. Something like *"A **home insurance** payment should be between 100 Finnish marks and 1000 Finnish marks when the **sales area** is other than Helsinki"*. Here the class variables are written in bold.

These rules are stored in relational tables having the rule name as the common id. This structure allows us to have most variable rules with a varying number of class variables and so on.

THE GENERATOR

There is an application, written in SAS/AF that the users can use to maintain the rule base. When the user has finished updating the rules for the material(s), he or she clicks a button and types the name of the material. The generator - also a SAS/AF-application - is started with the name of the material as its parameter. After the normal initializations, the generator reads a file of a few hundred lines of SAS program code and writes those lines to the beginning of the checking program to be. These lines consists of a set of macros to shorten the program and make it more flexible. Most of the rule code is implemented in the form of SAS macro code.

The structure of the rule base is somewhat complicated, and the generator should be able to read the rules regardless of the differences between the rules, then generate the code. After a lot of discussion it was decided to use SCL lists as the temporary structures for the rules. In this context the list concept SCL supports, will really shine. The generator opens a list for all the rules with each rule as a list item. Within this list there are sub-lists: each individual rule is a sub-list with all the data of the rule as list members. The information about the (variable number of) class variables will be stored in yet another sub-sub-list. The final structure seems complicated when drawn, even somewhat recursive, but it is extremely flexible and relatively easy and straightforward to implement.

The generator first builds the lists, then begins to **cdr** (according to LISP tradition: "walk along the list") down the lists. For each list element it writes out program code according to the list element. At this point the working of the generator resembles that of a compiler: having this sort of source expression, write out this piece of code. List processing is the key. Without the capability for lists, a generator would have been much more complicated and difficult to implement. After the rules have been processed, the generator writes more or less constant code to write error reports, update metadata tables (like the status table to contain the status of each material: *updated OK, updated with errors, not updated*). Finally the program will contain code to clean up libraries.

These generated checking programs will be run automagically after each input material has been read (collected from production databases). Normally there is no need for human involvement. This is the theory. In the case of errors we have to re-run the corresponding input program

sometime (as soon as either the errors have been corrected, or the rule base has been updated) and the checking program very soon thereafter. This calls for some manual intervention.

CONCLUSIONS

The use of a generator has made it possible to create a system that will check the information going into the DW as thoroughly as reasonably possible and to maintain some dozens of checking programs totally without DP personnel. All that is needed is a business expert to maintain the rules in the way he or she understands. The difficult parts were the design of the rule base and the generator. The more flexibility is needed in the rule base, the more complicated and difficult they become, so a line has to be drawn somewhere. In this case we have spent quite a lot of time in philosophical discussions about the structure and meaning of the rules. After a lot of design the final rule base is clear and understandable. As for the generator, the difficulty was more or less in the technical side. Some people find it difficult to understand the work and structure of a generator, as they may find it difficult to understand what exactly will happen when the generator is run and what will happen when the generated program is run. When this is understood and mastered, a generator is not overly complicated as a program. With a generator the program maintenance can be totally automated.

CHOICE OF LANGUAGE

The language available may pose some limitations in the program and data structures. In our case this is all done with SAS products. The DW is in the form of SAS files. Every single program used to get information in and out of the DW is written in SAS. The graphical user interface is written with SAS application facility, SAS/AF and programming language SCL. The generator also is written in SCL and the generated programs are SAS programs. SCL is a language which contains SAS language and some extra features. SCL was originally meant for building interactive SAS applications, which explains some added structures, the most interesting and important of which was the list processing capability.

TRADE MARKS

SAMPO INSURANCE COMPANY PLC

Sampo Insurance Company Plc is a part of Sampo, the largest provide of financial services in Finland. I work here as the local SAS Consultant and expert. We have had SAS since 1989. I have also been the chairman of SUGIF, SAS Users' Group In Finland for a number of years.

For further information, comment and discussion, please contact:

Markku Suni
Consultant
Sampo Insurance Company Plc
P.O.B. 2063
FIN-20075 SAMPO
Finland

Tel: +358-10-514-2095
Fax: +358-10-514-2124
e-mail: markku.suni @sampo.fi