Logic Programming to Generate Complex and Meaningful Test Data

Donald Maffly

dmaffly@huronconsultinggroup.com

Abstract

Even in today's modern software engineering world, there still exist obstacles that confound testers' ability to generate test data—data that is complex and precise but also that can be generated in mass quantity quickly.

In data warehouse applications, which typically contain copious historical data, it is very cumbersome and time consuming to generate data through the 'front door' of these systems via import files and simulated user interaction. This approach can also have the shortcoming of not easily allowing application testers to land data on precise points called for in functional testing.

Furthermore, third party data generation tools, while excelling at generating copious data, generally don't provide application testers customization hooks that are powerful and expressive enough to generate meaningful data — particularly data where relationships are complex and where understanding of the data requires application knowledge (e.g. business logic) not found in a database schema.

To address this problem, we have taken an approach using constraint logic programming; it is loosely inspired by prior work done in the area of Reverse Query Processing (RQP). The general idea is that if a tester can write a SQL SELECT query for the test data s/he targets, then using RQP the tester has the tool to generate data sets that fit that query.

Biography

Donald Maffly has over 25 years of experience in software development, not only as a tester, but also as a developer and manager; over the years, he has worked on a diverse range of technologies and products, ranging from those serving the financial, healthcare, and R&D communities. Currently at Huron Consulting Group for the past 5 years, Donald has focused entirely on a Software Quality Assurance role in the Business Intelligence and Data Analytics areas.

1 Introduction

If system architecture is the “bones” of a system, the software code is the “flesh” of a system, then data represents the “life blood” of a system. Data constantly flows through a system and it is constantly changing. It follows then that in the endeavor of testing the system, you need diverse means of getting vast and varied data into the system through all of its various inputs.

While there exist many ways to obtain test data, all of which can fill valid purposes, we found that there was something lacking in existing approaches that allowed us to practically, quickly get test data into the system to functionally test it.

We addressed this void by creating a tool that injects data directly into a database back end; it is loosely modeled after Reverse Query Processing (RQP) [BINNIG 07]. The idea is that if you can pinpoint the test data you need in the form a SQL query, then with an RQP tool you have the capability to generate data that satisfies that query.

We describe our system context and some of the challenges that we faced using existing approaches to obtaining test data. This sets the stage for deriving requirements for our tool, the design of the tool, and ultimately how it is used to generate test data.

2 Huron Consulting Group’s Data Generation Problem

To provide some context into the data generation problem we are confronted with at Huron Consulting Group, we describe the essentials of the system under test. The Aeos® application is a collaborative work flow system that supports the Huron process improvement methodologies for healthcare providers with an Online Analytical Processing (OLAP) back end database. The primary goal of the Aeos system is to get *more* patient-based revenue into the books *sooner*. Aeos software has two primary input points: (1) hospital-specific account/patient/billing information in import files from the client Hospital Information System (HIS) and (2) the Aeos application that hospital representatives use to work items to completion.

Data reaches the front end of the system as concrete entities such as account numbers, patient names, balances, and dates. By the time data reaches the data warehouse back end, the dimension of tracking change is added. A further data refinement occurs next when data warehouse Extract-Transform-Load (ETL) processes transform this data into more abstract forms such as counts and balances relative to dates. Finally, the end user sees the highly distilled final product as metric calculations in the form of data tables, charts and graphs. Business analysts and managers use these to assist in day-to-day management as well as to help make longer range strategic business decisions.

A hypothetical example of such a metric that we work through later in this document is “First Day Touched Done Percent”. This metric is the percentage of work items that are in the *done* state and that were edited or modified by a user on the first day it was assigned. The simple sounding name of this metric belies the true complexity of it. The data supporting it is quite complex, requiring conditional joins across many tables combined with additional filtering criteria.

Examples of data constructs that support the computation of these metrics are:

- Two-way linked lists. Our back end data warehouse contains many fact tables that contain self-referential two-way linked lists. These can be traversed forward or backward to track change of state. To further complicate things, these tables contain two sets of two-way linked lists: one to track same day change and another to track all-time change.
- Multiple date fields exist on multiple tables that semantically relate, but in ways that are not derivable from examination of the database schema alone.

Complications like these present different challenges depending on the approach we might choose to obtain our test data.

3 Shortfalls of existing data generation approaches

In this section, we summarize the approaches that we have considered to obtain complex test data. Each approach has its strengths and weaknesses, but, in the end, we concluded that they all fell short of our complex test data need. From here, we were able to describe precisely what our need was in the form of a concise list of requirements.

3.1 Where data harvested from production systems falls short

Data exists in bountiful quantity on production systems. Such data has the advantage that it is 'real' and it contains customized client specific properties that make the data truly unique. It is nearly impossible to mimic this production data in our test environments. For this reason, testing on client customized UAT systems is an indispensable part of our testing strategy. But it is not *central* to our testing strategy because working with this data comes with a few major obstructions.

1) Production data contains Personally Identifiable Information (PII). PII contains highly sensitive data such as social security numbers, account numbers, and other private information. In the health care arena, for instance, the Health Insurance Portability and Accountability Act (HIPAA) provides strict guidelines designed to protect this data (also referred to as Personal Health Information or PHI) and the privacy of individuals. Healthcare institutions can get fined for violating these rules. In other arenas, such as ecommerce, reputations of companies are at stake if PII is mishandled in any way. As a result, our clients (health care institutions) are generally reluctant to let their data migrate outside their network. So in cases where we do need production data to test with, we have no other recourse other than to test on client test systems over which we wield little control. We typically need to share these systems with other users (e.g., administrators, testers, business analysts), we are unable to install special software on them, and we are prohibited from performing 'destructive' type testing. This is far from an optimal environment in which to conduct testing.
2) Production data is *not* organized in a way that maps easily to our test cases; therefore it is nearly impossible to execute our entire test suite out on a client UAT system to validate it completely.
3) Production data is large and unwieldy. Consequently, it can lengthen testing cycles by an order of magnitude or more, and it is challenging to debug and diagnosis problems.
4) Production data doesn't necessarily land on all data points that are critical to functionally test the system in entirety.

3.2 Where conventionally loaded test data falls short

At Huron Consulting Group, we test on internal systems that contain a single canonical configuration that is intended to represent the many configurations that our clients have on their production systems. This canonical configuration captures most of clients' specifics but surely not all. We use a custom built in-house data generation tool that does a reasonably good job of generating production-like data into our system for testing purposes. This tool generates data the same way it is done in production — via import files as well as by mimicking end-user activity via the application. While the data that results is very realistic, it comes with a couple of drawbacks:

(1) The abstractions and language of our import files are very different than the abstractions that have been processed and land on the back end of our system in the data warehouse. This makes it challenging to script test cases using import file vocabulary that is targeting BI functionality expressed in an entirely different vocabulary. Thus, there is a cumbersome translation the tester needs to go through to script test cases with front end abstractions targeting back end functionality. If a test case requires user activity, it is 'hit or miss' using our in-house tool since it is not able to coordinate simulated end user activities on particular imported data. Ultimately we don't have the highest confidence that data is landing on precise points called out in our test cases.

(2) We need a lot of generated historical data to adequately test BI functionality since it delivers trend and weekly average information to the end user. Because our in-house tool imports data via conventional channels, it necessarily depends on the system clock to import data. Loading data this way is tedious and can take several days or longer.

Using our in-house tool to create test data in the back end data warehouse is akin to spraying a machine gun in a general direction to hit a target across a vast expanse. Not only are the bullets likely to miss precise targets, but it also takes bullets quite a while to arrive. At first blush, it would appear to be easier to shoot the target from point blank range. But that too, is not as easy as it would seem.

3.3 Where direct SQL data insertion falls short

When a front end data loading approach cannot meet our needs, an obvious fallback choice would be to code SQL INSERT statements to populate the data directly into the Aeos system's data warehouse back end, that is to say, shoot the target from point blank range. It turns out this SQL direct approach has proven quite useful in instances where it is sufficient to populate single tables (e.g. summary tables) in isolation of greater database context. However, this SQL direct approach falls short when creating complex networks of rows of data across many tables, if for no other reason than it is very complicated. After all, in live contexts, it requires a middle computing tier containing complex application logic to construct this data.

Furthermore, using a SQL direct approach, the test scripter loses greater relational context when he has to break his code into separate sections to perform SQL inserts across different tables. This has the effect of making the code more challenging to write, as well as to read and maintain. The approach also offers no obvious silver bullet to solving our problem of creating the two-way linked lists within single tables.

3.4 Where third-party data generation tools fall short

We searched for commercial off-the-shelf (COTS) solutions that could assist in injecting data directly into our BI database. We were sorely disappointed.

There exist numerous test data generation tools on the market that are all capable of generating copious amounts of test data. These tools provide end users with all kinds of tricks for generating data — randomly populated fields, sequentially populated fields, drawing from finite user defined domains and even offering end users the ultimate power of using SQL and Python hooks to generate column values.

While the resulting data generally suffices for certain types of testing (e.g. performance testing, testing UI front ends, etc.), it falls short for functionally testing BI data warehouses containing highly complex and abstract data. Using such tools it is very difficult to get data to land in cracks and crevices in a data profile that are meaningful as far as ETLs and metrics are concerned. If data happens to land in these meaningful places, it does so accidentally, not by design, even with the use of these tools' customization hooks.

We were able to find one or two data generation tools that could scratch the surface of generating meaningful data, but they do so in a clumsy non-intuitive manner. The best tool(s) allow users to script custom data generation from the perspective of a single table column, but not from the greater picture that a SQL query would provide, for instance, where relations between tables and constraints are immediately obvious to the coder or anyone else needing to understand the code. Using these tools, custom generated code is broken up into bits and pieces on a table column basis, and the greater context of the intended semantic can be lost.

3.5 Requirements for generating complex test data

From this problem exploration, we were able to derive the following requirements needed in a data generation tool:

1) Generate ‘smart’ data that is business application aware. Like COTS, the database schema can be leveraged. Unlike COTS, better customization hooks are needed to capture higher level business application logic.
2) Generate data that targets precise points in the data warehouse. These precise data points may have complex inter-table dependencies.
3) Generate data using customization hooks that are not myopically confined to just column value generation, but that allow the scripter greater vision of broader relational context in a manner that might not be all that different from the SQL language.
4) Generate historic data relatively quickly (at least an order of magnitude faster than our in-house data generation tool).

4 From Requirements to Solution

Given the limitations with commercial off-the-shelf tools, we fell back on our in-house generation tool despite its long and tedious data generation process. Afterward, we’d validate that we had the data we needed before initiating the actual testing. It is worth noting that the intention of these data validation scripts was not to test BI functionality of our product per se, but rather to see if the data was present to perform a given test.

We came up with a few hundred such scripts, each mapping to a particular test scenario targeting ETLs, all written in SQL. Even though each of these SQL queries generally spanned a handful of large tables (some with over 100 columns) in our data warehouse, each query was concerned with only 10 to 15 total column values across several tables, mirroring the ETL computation for which it was intended to validate test data. It was as if the remaining columns were invisible as far as the ETL was concerned. So any solution to generate test data for them would not need to concern itself with populating *complete* data, but rather *minimally sufficient* data. When approached this way, the test data generation problem becomes less daunting.

It occurred to us sometime during the laborious process of writing these SQL validation scripts that it would be much easier if we could simply directly generate the test data that these SQL scripts were querying. If only there existed a tool that could perform ‘reverse-sql’, as it were — not a SQL SELECT, but the opposite of it. It turns out that there has been research done in this area [BINNIG 07]. In their paper, Reverse Query Processing (RQP) the authors cite as follows:

> “*Reverse Query Processing (RQP) gets a query and a result as input and returns a possible database instance that could have produced that result for the query. […] There are several applications for RQP; most notably, testing database applications […]*”

This work never made it farther than the research phase, and no commercially available tool ever resulted from this work that we know about. We considered implementing RQP based on [BINNIG 07], but felt it would be difficult to translate into a practical tool. Our work presents a Constraint Logic Programming (CLP) based alternative to [BINNIG 07] which demonstrates RQP’s usefulness in commercial application.

4.1 Prolog as a platform for generating data

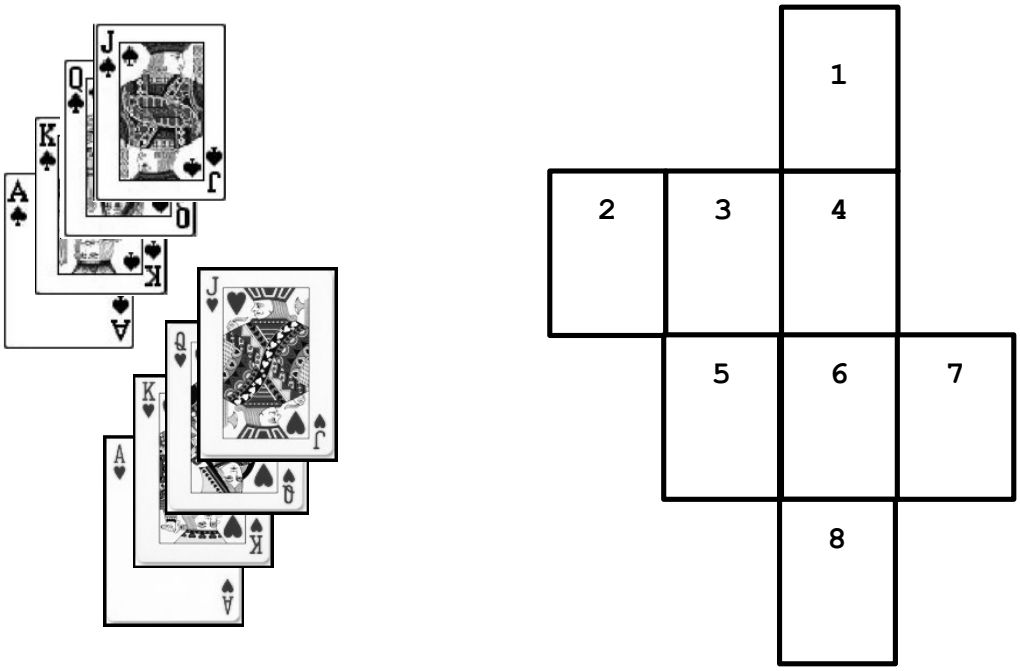
Prolog [KOWALSKI 74], [COLMERAUER 93] promised to be a suitable platform from which we could solve our data generation problem. Like SQL itself, Prolog is a declarative language, and it is well suited for expressing relationships between objects. Furthermore, the *list* is a data structure that is central to the Prolog language, Prolog would likely assist in the generation of Aeos software’s complex table data containing two-way linked lists. Finally, Prolog’s logic programming and rules capability makes it very powerful at generating *many* solutions to a given problem. If we could only figure out how to express our problem in a data generation-like vocabulary, a Prolog solution to the problem could be the relational data we sought.

For the sake of illustration, we walk you through how Prolog can be used to solve a simple logic problem, and then we extend this problem solving model to the greater problem of data generation.

4.1.1 Solving a simple logic problem with Prolog

There is a logic problem in [SUMMERS 72] called "Card Number Six". Given a grid of 8 squares (Figure 1), you need to distribute two aces, two kings, two queens, and two jacks on the grid such that:

- Two cards with the same denomination (e.g. 2 aces) may not border each other
- Each ace borders a king
- Each king borders a queen
- Each queen borders a jack
- No queen borders an ace



[Figure 1]

Think about how you might solve this in the most brute force way possible if you had limitless brain power at your disposal. Perhaps the simplest strategy would be to take all the combinations of the 8 cards across the grid and test each combination against our criteria listed above and see if it passes muster.

In Prolog, let's begin with our primary data structure, an ordered list of eight cards. The card in the first position of this list indicates that the card is on the first square on the grid, in the 2nd position, on the 2nd square, etc.:

```
CardList = [a1,a2,k1,k2,q1,q2,j1,j2]
```

Here, a1 denotes Ace One (i.e. Ace of Spades), a2 denotes Ace Two (i.e. Ace of Hearts), etc.

There is a built-in predicate in Prolog that is able to return all the permutations of this list:

```
permutation(CardList, PermutedList)
```

To run each permutation against our above listed criteria, we create some building blocks. First we code some Prolog facts that describe whether squares in the grid border each other or not:

```
borders(1,[4]).
borders(2,[3]).
…
borders(6,[4,5,7,8]).
```

…then two predicates `is_next_to` and `is_not_next_to` (whose body/definitions are not shown here) built on top of our **`borders`** facts from which we can express our filtering criteria:

```
is_next_to(a1,[k1,k2], PermutedList),
is_next_to(a2,[k1,k2], PermutedList),
is_next_to(k1,[q1,q2], PermutedList),
is_next_to(k2,[q1,q2], PermutedList),
is_next_to(q1,[j1,j2], PermutedList),
is_next_to(q2,[j1,j2], PermutedList),
is_not_next_to(q1,[a1,a2], PermutedList),
is_not_next_to(q2,[a1,a2], PermutedList),
is_not_next_to(a1,[a2], PermutedList),
is_not_next_to(a2,[a1], PermutedList),
is_not_next_to(k1,[k2], PermutedList),
is_not_next_to(k2,[k1], PermutedList),
is_not_next_to(q1,[q2], PermutedList),
is_not_next_to(q2,[q1], PermutedList),
is_not_next_to(j1,[j2], PermutedList),
is_not_next_to(j2,[j1], PermutedList).
```

Prolog returns not just the cards that could land on square number 6, but *all* the squares, as follows:

```
X = [k1, q1, j1, q2, a1, k2, j2, a2];
X = [k1, q1, j1, q2, a1, k2, a2, j2];
X = [k1, q1, j1, q2, a2, k2, j2, a1];
X = [k1, q1, j1, q2, a2, k2, a1, j2];
…
X = [k2, q2, j2, q1, a1, k1, j1, a2];
X = [k2, q2, j2, q1, a1, k1, a2, j1];
X = [k2, q2, j2, q1, a2, k1, j1, a1];
X = [k2, q2, j2, q1, a2, k1, a1, j1].
```

Not all solutions are listed here. For the sake of brevity, only 8 out of the 32 possible solutions are listed. Note that only a king can land in the 6th position, i.e., either the King of Hearts or the King of Spades.

4.1.2 From 'Card Number Six' to data generation

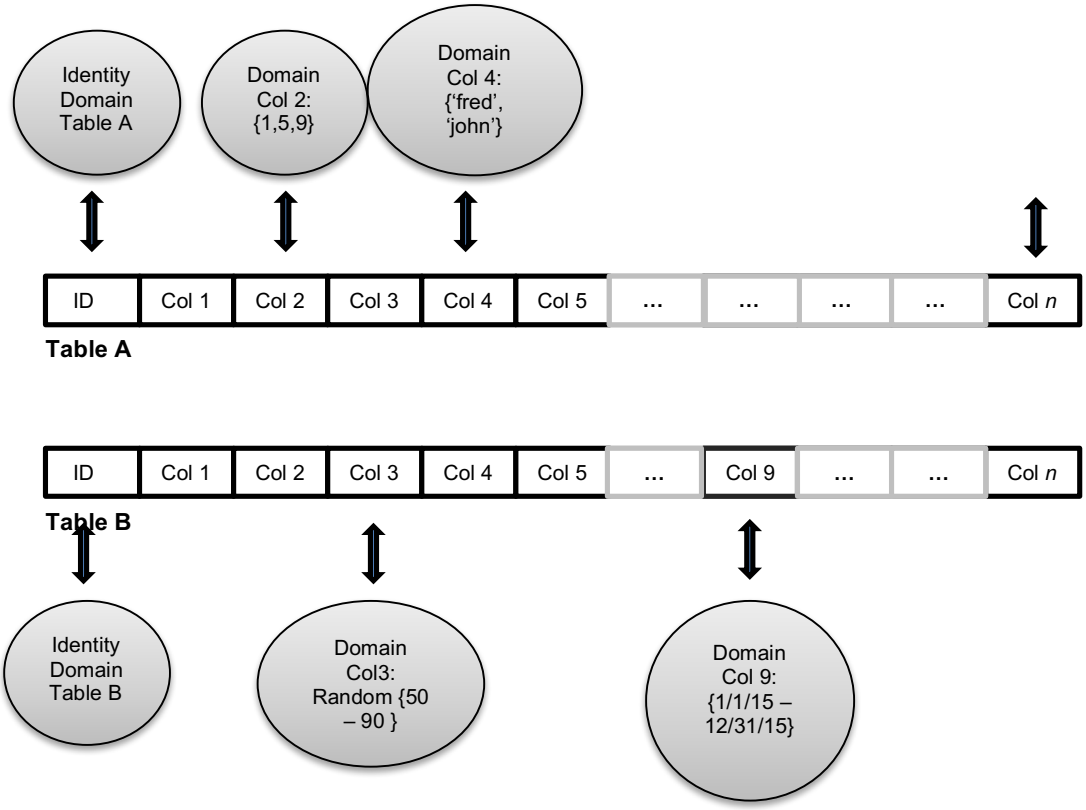
It is not a huge leap to extend our approach to solving the "Card Number Six" problem to the arena of data generation with a few minor tweaks. Imagine instead that the grid in our card problem was a row of data in a table, each square being a column value. Then imagine that only one of four queens (heart, spade, club, and diamond) were allowed to land on square one, kings on square two, etc. The programmer could modify the criteria to ones that were appropriate for a particular test case; e.g. produce all the combinations where only cards with denomination of hearts are contained in the grid, or the row of data as it were.

Our data generation tool follows this same approach to generate complex data scenarios:

1) Create permutations of table row data

2) Throw out the meaningless permutations by evaluating each against test specific criteria

Recall in our “Card Number Six” approach, we didn’t employ a particularly smart and efficient problem solving strategy, but it’s worth noting that the complexity of the problem and the small size of the search space did not necessitate it. The initial set of permutations numbered 40,320 possibilities, large for the human mind to grapple with, but quite manageable for your average laptop computer to handle. In the case of more complex problems with larger search spaces, such as the data generation problem, search spaces can grow several orders of magnitude larger and can easily exhaust the memory and computing resources of a machine.



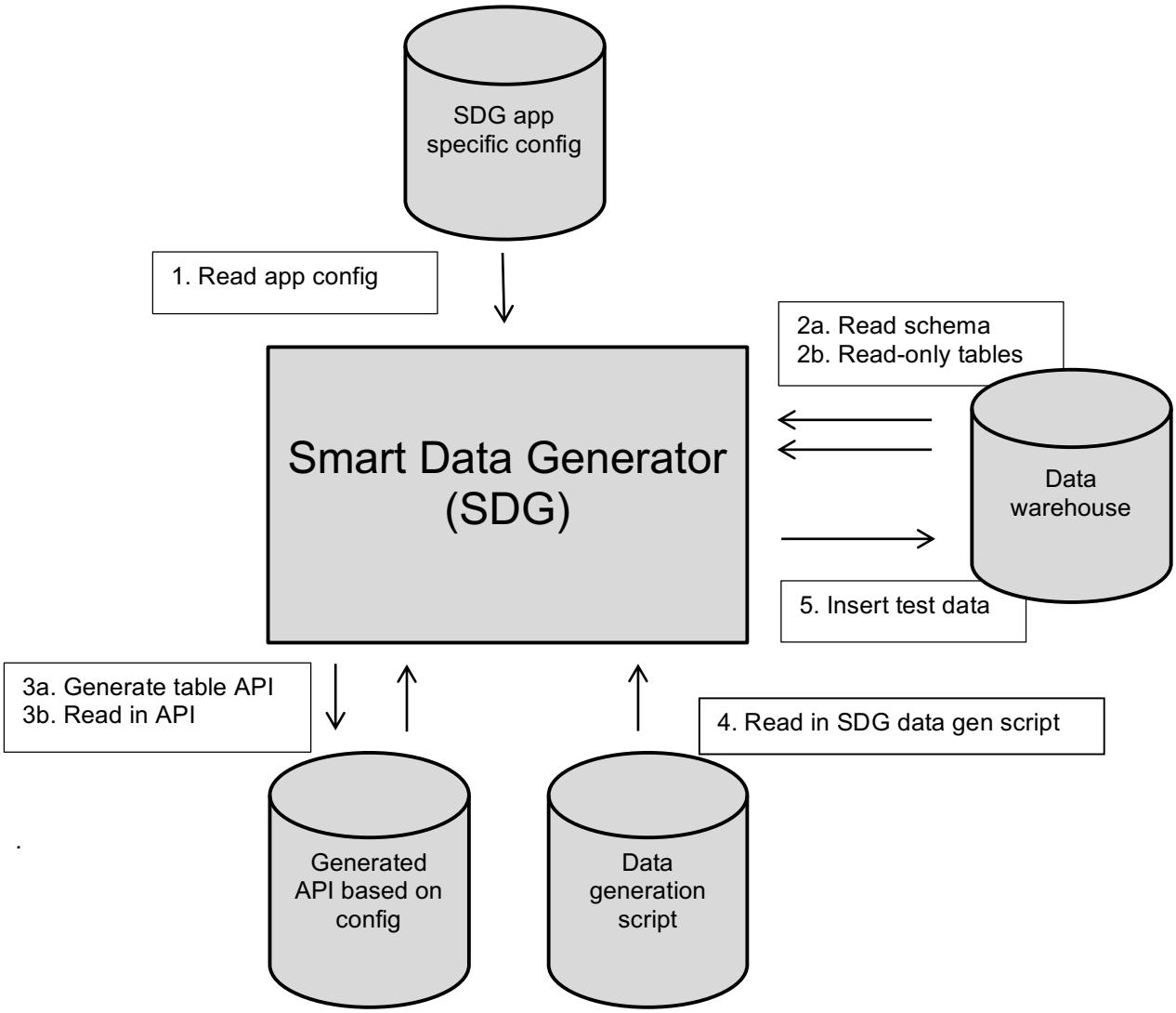
[Figure 2]

Our tool’s approach to overcoming this limitation is to allow the test scripter to constrain the pool of possibilities from which values for a particular column can be drawn *prior* to generating data combinations. These are referred to as finite domains in the logic programming community; it is one of several critical extensions to Prolog to support ‘constraint logic programming’ [JAFFAR 94]. While some dialects of Prolog provide programmers the ability to create finite domains for logic programming using integers [TRISKA 14], our tool has specialized finite domain machinery that is more suitable for the data generation problem.

Figure 2 depicts a simple example data scenario using finite domains across two different tables, A and B. We define finite domains for columns that are meaningful as far as a particular test scenario is concerned. Assume that the remaining columns are irrelevant to this particular test scenario. Columns are given either system or project defined single default values that are generally workable. Keeping the size of these domains in check has the effect of keeping the search space manageable. Generally, the number of data scenarios that can be computed as the Cartesian product of the size of all of the domains. In our example above, we end up with (1 x 3 x 2) x (1 x 1 x 365) = 2190 distinct data scenarios.

5 SDG: RQP powered by Prolog

We developed our test data generator using the problem solving strategy used in the “Card Number Six” example. We have initially dubbed it ‘Smart Data Generator’ (SDG) — patent pending. We chose SWI-Prolog [SWI-PROLOG 12] as the platform on which SDG runs. Attractive SWI-Prolog traits are its broad supporting library infrastructure, an active user community and a friendly licensing policy. However, SDG is theoretically capable of running on any Prolog platform since it is coded to standard Prolog APIs. An architectural overview of SDG is depicted in Figure 3.



.

[Figure 3]

SDG reads in an application configuration file (#1, in Figure 3), which lists the tables (both read-only and write) needed to generate test data. This configuration file optionally can contain column specific information such as default values and column aliases. Using this configuration, SDG reads table and column schema information from the database (#2) that will be referenced in the creation of data. SDG then generates an API to these tables (#3), in addition to some ”under the hood” machinery for handling *finite domains* on each table column. Generating database APIs from relational models is a well

understood and solvable problem [ORM], [PERSISTENCE], [HIBERNATE]. SDG follows this same approach to generate APIs for each table as follows:

- Row object creators/constructors for each table
- Getters for each column on the table
- Setters for each column on the table

In step (#4), SDG processes test specific data generation scripts coded to the aforementioned generated API, Prolog's unification and back tracking inference engine does the lion's share of the work from there to generate data set solutions, and finally (#5) SDG inserts data directly into the database.

Referring back to our example of two tables in the prior section, we walk through a very concrete example, albeit fictional, of a scripted test case, and demonstrate how SDG would process it. For the sake of illustration, we'll simplify and remove the primary key from the example; given table T, with columns x and y.

```
assertDomain(T.x, [1,2]),
assertDomain(T.y,[a,b]).
```

Here we define the possible values that `T.x` can be to be 1 and 2; `T.y` to be a and b. Prolog will generate 2 x 2 (4) solutions for this goal, `create_T(T)`:

```
T=(1,a);
T=(1,b);
T=(2,a);
T=(2,b).
```

Adding a primary key identity to T will not increase the number of permutations; we add it in our next example adding a 2nd table T2 with foreign key to T, to which we will 'reverse' equijoin. Say T now has additional column oid (unique increment) and table T2 is containing columns oid,x,y,fkToT.

```
assertDomain(T2.X, [1,2]),
assertDomain(T2.Y,[a,b]).
```

The Prolog goal:

```
create_T(T), create_T2(T2),set_T2_fkToT(T2, T.oid).
```

will yield the following (2 x 2 x 2 x 2) 16 solutions:

```
T=(1,1,a), T2=(1,1,a,1);
T=(2,1,b), T2=(2,1,a,2);
T=(3,2,a), T2=(3,1,a,3);
T=(4,2,b), T2=(4,1,a,4);
T=(5,1,a), T2=(5,2,a,5);
T=(6,1,b), T2=(6,2,a,6);
T=(7,2,a), T2=(7,2,a,7);
T=(8,2,b), T2=(8,2,a,8);
T=(9,1,a), T2=(9,2,b,9);
T=(10,1,b), T2=(10,2,b,10);
T=(11,2,a), T2=(11,2,b,11);
T=(12,2,b), T2=(12,2,b,12);
T=(13,1,a), T2=(13,1,b,13);
T=(14,1,b), T2=(14,1,b,14);
T=(15,2,a), T2=(15,1,b,15);
T=(16,2,b), T2=(16,1,b,16);
```

Every solution is separated by semi-colons above and constitutes a separate data set. Let's finally demonstrate the addition of some constraints to filter out unwanted permutations. For example, we want to discard any data set where T1.x = T2.x. In bold italics, we add to our goal:

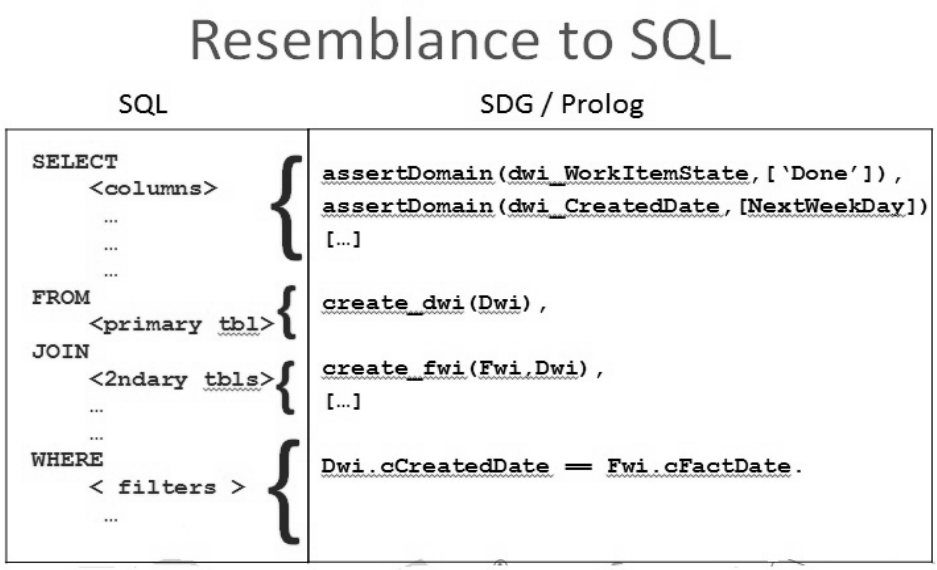
```
create_T(T), create_T2(T2),set_T2_fkToT(T2, T.oid), T.x \== T2.x.
```

This has the effect of culling out the number of solutions (data sets) by 8 :

```
~~T=(1,1,a), T2=(1,1,a,1);~~
~~T=(2,1,b), T2=(2,1,a,2);~~
T=(3,2,a), T2=(3,1,a,3);
T=(4,2,b), T2=(4,1,a,4);
T=(5,1,a), T2=(5,2,a,5);
T=(6,1,b), T2=(6,2,a,6);
~~T=(7,2,a), T2=(7,2,a,7);~~
~~T=(8,2,b), T2=(8,2,a,8);~~
T=(9,1,a), T2=(9,2,b,9);
T=(10,1,b), T2=(10,2,b,10);
~~T=(11,2,a), T2=(11,2,b,11);~~
~~T=(12,2,b), T2=(12,2,b,12);~~
~~T=(13,1,a), T2=(13,1,b,13);~~
~~T=(14,1,b), T2=(14,1,b,14);~~
T=(15,2,a), T2=(15,1,b,15);
T=(16,2,b), T2=(16,1,b,16);
```

6 SDG as a realization of Reverse Query Processing

For the intrepid reader, we provide an in depth example using SDG to generate data to test a data warehouse ETL in appendix A1. The biggest take away with this example is that the SDG script closely resembles the SQL query in structure, and that the semantic of the SDG is essentially the reverse (Reverse Query Processing if you will) of the SQL semantic. Figure 4 makes this similarity readily apparent by juxtaposing SQL side by side with the SDG / Prolog code:



[Figure 4]

The `assertDomain` calls essentially define the column values that a SQL SELECT statement would return. The generated `create_XXX` calls are analogous to SQL JOINs. Finally, any trailing filters correspond to a SQL WHERE clause.

While the SDG does not generate *all* the data sets that could qualify for a SQL query, using finite domains, the SDG scripter has the tool he needs to limit the number of data sets returned — particularly down to those that are meaningful as far as a test case is concerned. Do realize that *all* the possible data sets could potentially be prohibitively large and no doubt contain massive redundancy as far as testing value is concerned. Using tried and true testing techniques such as combinatorial pairwise testing [KUHN 10], the SDG scripter can further pare down the number of data sets returned to gain the greatest testing value for a given minimal amount of data.

7 Closing Remarks

We have developed and utilized our SDG tool to test the OLAP back end of the Aeos system over the last year. We have found that it has proven highly effective at generating complex data used to functionally test complicated ETL processes used in the process of delivering business intelligence to end users. In fact, SDG has met all the other requirements that we called out originally. It requires users to script in a language that should be plainly intuitive to SQL programmers, and it also enables the easy derivation of data generation scripts from SQL where that is needed. We have also been able to inject data into the back end of our system a couple of orders magnitude faster than with our original in-house data generation tool.

Our approach can likely be generalized, from generating data to test an OLAP application to any scenario where relational data is sufficiently complex that it requires deep understanding of higher level business application logic.

While in truth, SDG is not a reverse SQL engine per se, there is no denying that a SDG script bears a very strong resemblance to SQL in structure and semantic. Our ETL testing example demonstrates that an SDG script is essentially a SQL SELECT in reverse, i.e. reverse query processing. So far, our use of SDG has been confined to mirroring relatively simple SQL statements. Further exploration will be required to see if SDG is up to the challenge of mirroring complex SQL statements containing, for example, OUTER JOIN, UNION, or COALESCE constructs.

An unanticipated result we found with SDG is that it does well to abstract away the often messy proposition of communicating with a relational database (e.g., order of inserts, complex SQL insert syntax, locks, optimization, etc.). Consequently, SDG scripts are more succinct, readable and maintainable than would be compared to writing SQL directly. Ultimately SDG allows the test scripter to focus on what is important, creating test data, without the distracting tedium of how to get data into the database. This is a generally recognized benefit that comes with any object relational technology [ORM].

Finally, in the interest of keeping the focus of this paper on Prolog as a test data generator, we have either glossed over highly detailed aspects of SDG or omitted mention of them completely. Additional details can be found in Appendix A2. Also, we made mention of the existence of two-way linked lists embedded in some of our tables, but did not discuss how SDG addresses the problem of generating this data construct; the reader should refer to Appendix A3 to gain deeper understanding.

8 Future Work

Since SDG is capable of generating data in CSV format, it could very easily be extended to supply data in scenarios where *any* data driven testing (DDT) methodology is being used, irrespective of the presence of a relational database. SDG could have even broader reach if it were able to generate into other data formats (e.g. XML) as well.

SDG is built leveraging the Prolog platform. However, other logic programming platforms could be considered. While we have not yet investigated Datalog [CERI 89], we suspect that it might provide a tighter integration with databases than do most Prolog platforms. Like Prolog, Datalog provides some of the same strengths that come with logic programming environments, but in addition to that, it also provides built in caching abilities so that data could be read in from the database without blowing up memory. Datalog might also provide improved database write functionality and performance. This could prove to be useful in scenarios where generating massive data was called for, as well as considering the state of data pre-existing in the database in the process of generating new data.

Although the SDG tool was originally conceived as a test data generator, because it provides an API to relational data, it could no doubt be extended to serve as middle ware in any multi-tiered application. SDG's logic programming capabilities could provide a whole new frontier that we don't typically see in multi-tiered applications. Furthermore, these features also make SDG a promising data analytics platform.

Glossary

Aeos® ***software*** – Huron Consulting Group's OLAP/OLTP software solution to assist hospitals in bringing revenue into the books

BI – Business Intelligence

Constraint logic programming (CLP) – a refinement in logic programming to enable logic programs to operate on data types that might otherwise have virtually infinite domains and thus blow up search spaces

data warehouse – a back end data store architected in a 'star configuration' to support business intelligence and data analytics; contains fact records, dim records and summary table data

dim records – detailed data in a data warehouse that is generally static in nature and constitutes dimensional axes in data warehouse star architectures; contrast with fact records

ETL – acronym for 'Extract, Transform and Load'. In Aeos context, this means refining detail data into summary data via aggregate computation

fact records – detail data in a data warehouse that is more dynamic in nature, generally capturing deltas in data over time; contrast with dim records

finite domain – an important component in constraint logic programming

measure – an aggregate in a summary table computed via ETL from detail data (fact records and dim records) in a data warehouse

metric – a computation built on summary table measures, which has business meaning. It is intended to support business stakeholders in business intelligence and data analytics activities.

OLAP – Online Analytical Processing

OLTP – Online Transaction Processing

SQL – abbreviation for 'Structured Query Language', the de facto language of choice for the retrieval and manipulation of relational data

Appendix

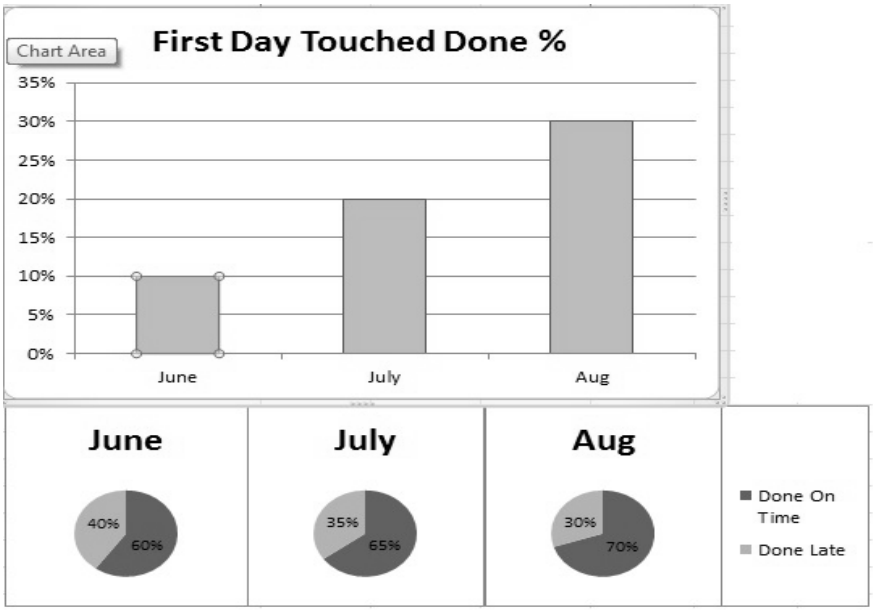
A1. Using SDG to generate data to test an ETL

For the sake of protecting Huron Consulting Group's sensitive intellectual property, we do not give an example from testing Aeos software, but rather we concoct a fictional scenario — one that is simpler, and one with which the reader may be more familiar. Team Foundation Server (TFS), a Microsoft product, is a tool that you may have heard of or used before, particularly if you are involved in agile software development on the Windows platform. Like Aeos software, TFS is a collaborative workflow tool. *[If this example bears any resemblance to any software analytics or business intelligence solutions (e.g. [CODEMINE 13] associated with Microsoft products, it is purely coincidental and not intentional.*]

Imagine now that there happens to be a business intelligence back end to TFS, and let's say data scientists have observed that when a work item has been touched by the assignee the very same day that it was first assigned to him/her, there was not only a higher likelihood the work item would reach a *done* state by the due date, but also the quality of the work would generally be higher. So business analysts figured it would be advantageous to create a metric and track TFS users' behavior over time, and finally report on it to managers who would then encourage his/her team to get an early start on their work. If the data analytics were right, the team would get higher percentage of work done on time.

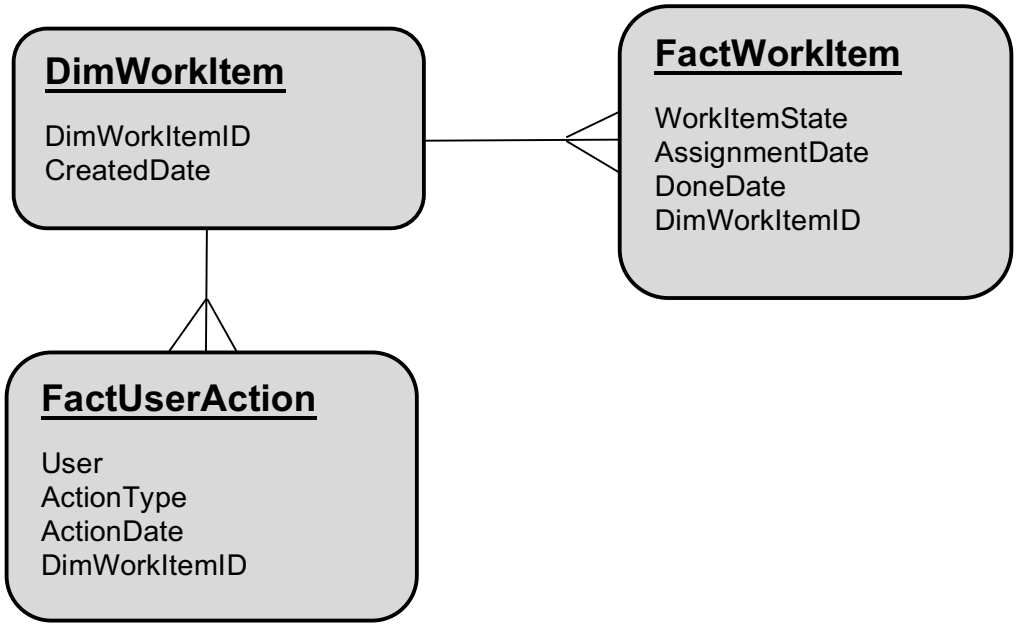
Let's call this metric "First Day Touched Done Percent" to be reported on a monthly basis across TFS teams and work item types. This metric requires a numerator summary measure *WorkItemsFirstDayTouchDoneCount,* which is a count of work items Done in a particular month, and were touched by the assignee the first day of assignment (not necessarily within the month). The metric also requires a denominator summary measure *WorkItemsDoneCount,* which is the total count of work items Done in the last month *regardless* of when the work item was first touched by the assignee after assignment. Both of these summary measures are aggregated across teams and work item types to be computed using an ETL processing 'detail' data.

This metric will be displayed to managers in an intuitive graphical format that will display "First Day Touched Done Percent" side by side with Done Late Vs Done On Time pie charts, that might demonstrate that as the percentage of First Day Touched work items increases, so does the work that is done on time. See graphical metric display in Figure 5.



[Figure 5]

Detail data DimWorkItem table contains status information about the work item, such as work item type and created date. FactWorkItem table tracks changes on work items, containing information about what the state change was and when the state change occurred. The FactUserAction describes the actions that occurred on work items. It would contain the type of the action as well as when the action occurred and who performed the action. Refer to E-R Diagram in Figure 6.



[Figure 6]

We take a simple 'divide and conquer' approach to devising test data. Generate data that qualifies for the measures and data that does not qualify. We'll call these 'data cases' and we separate them into these 4 distinct categories:

1. Qualify for *WorkItemDoneCount* : We need work items that transitioned into Done state during the reporting week. Work items need be assigned to users on and after the CreatedDate and both on and before the DoneDate. Work items need to be assigned to users that span different teams. Work items should be of different types. Work items should have their Done Date coming before, on and after the DueDate.
2. Qualify for *WorkItemFirstDayTouchDoneCount:* same as for *WorkItemDoneCount* with the added constraint that the work item has a UserAction on the same date as the AssignmentDate.

3. Not qualify for *WorkItemDoneCount*: (1) Work Item is not Done or (2) DoneDate is not in the reporting week (i.e. not between first day of the week and the last day of the week), (3) work items to be assigned to different users on different teams; work items to span different work item types, work items whose Done Date comes before, on and after the DueDate.Not qualify for *WorkItemDoneCount*: (1) Work Item is not Done or (2) DoneDate is not in the reporting week (i.e. not between first day of the week and the last day of the week), (3) work items to be assigned to different users on different teams; work items to span different work item types, work items whose Done Date comes before, on and after the DueDate.

4. Not qualify for *WorkItemFirstDayTouchDoneCount:* same as for *WorkItemDoneCount* with the added constraint that the work item has a UserAction NOT on the same date as the AssignmentDate.

For the purpose of demonstration, let's focus just on data case 4. We'll first create a SQL SELECT query for this, and from there we will port it into SDG/Prolog to generate data that would qualify for this SQL query.

```
SELECT DWI.DimWorkItemID,
       DWI.Type,
       FWI.DoneDate,
       FWI.AssignmentDate,
       FUA.ActionDate AS FirstTouchDate
FROM DimWorkItem DWI
JOIN FactWorkItem FWI
 ON FWI.DimWorkItemID = DWI.DimWorkItemID
 AND FWI.DoneDate BETWEEN @Monday AND @Friday
    AND FWI.DueDate = @Tuesday
JOIN FactUserAction FUA
 ON FUA.DimWorkItemID = DWI.DimWorkItemID
 AND FUA.ActionDate > FWI.AssignmentDate
WHERE DWI.WorkItemState = 'Done'
AND DWI.CreatedDate < @Monday
AND DWI.Type IN ('SBI', 'PBI', 'BUG')
AND FactUserAction.User IN ('Viji', 'John', 'Hans')
```

In this data case, everything qualifies the data for the *WorkItemFirstDayTouchDoneCount* measure except for the line in bold italics.

The equivalent SDG/Prolog script to generate data sets for this is as follows:

```
assertDomain(dwi_WorkItemState, ['Done']),
assertDomain(dwi_CreatedDate, [LastWeekDay]),
assertDomain(dwi_Type, ['SBI', 'PBI', 'BUG']),
assertDomain(fwi_AssignmentDate, [Monday]),
assertDomain(fwi_DueDate, [Tuesday]),
assertDomainRange(fua_ActionDate, Tuesday, Friday),
assertDomain(fua_user, ['Viji', 'John', 'Hans']),
findall(([Dwi,Fwi,Fua],
      (  create_dwi(Dwi),
         create_fwi(Fwi,Dwi),
         create_fua(Fua,Dwi)), DataSets),
sqlInsert(dataSets).
```

In this example, 'LastMonthDay', 'Monday', 'Tuesday', 'Friday' are variables that are set globally and adjusted per the reporting month that the tester is targeting. Note that with a simple change to the line in bold italics, we can change this script from one that generates non-qualifying data to one that generates qualifying data for our measure. By changing 'Tuesday' to 'Monday', data will be generated where the ActionDate indeed equals the AssignmentDate, (data case #2).

A2. SDG in Finer Detail

1) **Expressibility in defining finite domains on a column**. Following the lead of third party data generation tools, SDG provides many options for creating domains on a given column, e.g.

random N from a set of values, finite domains into read only tables either directly or symbolically, finite domains as ranges, finite domains via Prolog query, etc.

2) **Read-only tables (i.e., tables for which data will not be generated).** SDG recognizes the read-only table abstraction and it fits nicely with our *finite domain* machinery. Scripters are free to set domains for column values that are actually foreign keys into read-only tables (typically containing dimensional data). These read-only tables can be referenced in script code symbolically (e.g. via code or English description) as long as there exists a designated column in that read-only table that can act as a symbolic identifier (e.g. 'John Smith', 'Legal Department', etc.).
3) **Triggers and data values that are dependent on other data values**. We have not given much attention to generic reusable trigger-like mechanisms. These would surely be useful, but we have been able to get by with setting dependent column values on a case by case basis as they arise. Furthermore, there is nothing to prevent an SDG scripter from writing customized Prolog predicates to address inter or intra table data dependencies in an application specific manner.
4) **Representation of primary and foreign keys**. In tables for which SDG generates data, it initially maintains all primary and foreign keys as object references just up until the time that they are inserted into the database. At this point, these object references are converted into object ids (OID) as they are represented in the database. We have taken this approach primarily because a lot of these data sets filtered out depending on the data generation script, and it is wasteful and problematic to give them OID keys prematurely.
5) **Tables with composite primary keys**. The Aeos data warehouse application has an overwhelming majority of its tables whose primary key is defined as a single unique OID, so we were not immediately compelled to grapple with the problem of composite primary key data generation. In the few cases where we do have tables with composite primary keys, we were careful in our data generation scripts NOT to generate more than one row with the same composite key. Certainly, there is possibility for improved support in SDG in this area.

A3. Table Data Two-way Linked Lists

This snippet of Prolog code creates a two way linked list of FactWorkItem rows; see ER diagram in appendix A1. Added columns are `FactWorkItemID` primary key, and `NextId`, `PrevID` as our list pointers. Prolog's built-in list data structure and the use of recursion help make this code very succinct.

```
create_fwi_chain_(1, _, ChainIn, ChainOut) :-
    ChainOut = ChainIn,!.
create_fwi_chain_(Len, LastFact, ChainIn, ChainOut) :-
    NewLen is Len - 1,
    create_fwi(F1),append(ChainIn, [F1], ChainTemp),
    get_fwi_FactWorkItemID(F1, Pk_F1),get_fwi_FactWorkItemID(LastFact,Pk_LastFact),
    set_fwi_NextID(LastFact, Pk_F1),
    set_fwi_PrevID(F1, Pk_LastFact),
    set_fwi_NextID(F1, null),
    create_fwi_chain_(NewLen, F1, ChainTemp, ChainOut).
create_fwi_chain_main(Len, ChainOut) :-
    create_fwi(F1),
    set_fwi_PrevID(F1, null),set_fwi_NextID(F1, null),
    create_fwi_chain_(Len, F1, [F1], ChainOut),!.
```

References

[BINNIG 07] C Binnig, D. Kossmann, E. Lo. *Reverse Query Processing*. Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on Data Engineering.

[CERI 89] S Ceri, G Gottlob, L Tanca; *What You Always Wanted to Know About Datalog*; IEEE Transactions on Knowledge and Data Engineering, Vol 1. No 1, March 1989

[CODEMINE 13], J. Czerwonka, N. Nagappan, W Schulte, B. Murphy, *CODEMINE: Building a Software Development Data Analytics Platform at Microsoft*, IEEE Software, Issue No. 4 – July Aug (2013 vol 30), pp. 64-71

[COLMERAUER 93] A. Colmerauer, P. Roussel. *The Birth of Prolog*, ACM SIGPLAN Notices, 1993.

[HIBERNATE] Hibernate. https://en.wikipedia.org/wiki/Hibernate_(Java)

[JAFFAR 94] J Jaffar, M Maher; *Constraint Language Programming: A Survey*, Journal of Logic Programming 1994: 19,20: pp 503-581.

[KUHN 10] D Kohn, R. Kacker, Y Lei, *Practical Combinatorial Testing*, NIST Special Publication 800-142, October 2010

[KOWALSKI 74] R Kowalski. *Predicate Logic as Programming Language*, Proceedings IFIP Congress, Stockholm, North Holland Publishing Co. 1974.

[ORM] *Object Relational Mapping*, https://en.wikipedia.org/wiki/Object-relational_mapping

[PERSISTENCE] Persistence Software. https://en.wikipedia.org/wiki/Persistence_Software

[SUMMERS 72] George J Summers; *Test Your Logic: 50 Puzzles in Deductive Reasoning*, Dover Publications Inc., New York, 1972.

[SWI-PROLOG 12] Wielemaker, Jan and Schrijvers, Tom and Triska, Markus and Lager, Torbj, *SWI-Prolog*, Theory and Practice of Logic Programming, vol 12, no. 1-2 p67-98, 2012

[TRISKA 14] Markus Triska, *Correctness Considerations in CLP(FD) Systems*, Vienna University of Technology, 2014

Acknowledgements

Many thanks to Huron Consulting Group for creating an environment that fosters a culture of innovation, without which this work would not be possible. Thanks to Huron Consulting Group colleagues Eswara Jadda , Viji Ramadoss, Yogesh Lulla, Joe Woolston and Rob Weight for paying a critical eye to this work and profoundly shaping its direction. Special thanks to referees Patt Thomasson and Dave Patterson for their thorough review of this document to transform it into a polished and finished product; to Anne Ogborn of the SWI-Prolog community for validating SDG's use of Prolog; to Chris Keene and Derek Henninger for vetting this work from the perspective of object relational mapping (ORM).