SYNTHETIC DATA GENERATION:
THEORY, TECHNIQUES AND APPLICATIONS
SYNTHETIC DATA GENERATION: THEORY, TECHNIQUES AND APPLICATIONS

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

By

Joseph E. Hoag
M.S. in Computer Science
University of California at Davis, 1991

December 2007
University of Arkansas
ABSTRACT

The need for synthetically generated data is growing rapidly as the size of enterprise applications increases. Situations requiring this technology include regression testing of database applications, data mining applications, and the need to supply “realistic but not real” data for third party application development. The common approach today to supplying this need involves the manual creation of special-purpose data generators for specific data sets. This dissertation describes a general purpose synthetic data generation framework. Such a framework significantly speeds up the process of describing and generating synthetic data. The framework includes a language called SDDL that is capable of describing complex data sets and a generation engine called SDG which supports parallel data generation. Related theory in the areas of the relational model, E-R diagrams, randomness and data obfuscation is explored. Finally, the power and flexibility of the SDG/SDDL framework are demonstrated by applying the framework to a collection of applications.
This dissertation is approved for recommendation to the Graduate Council.

Dissertation Director:

____________________________________
Craig W. Thompson

Dissertation Committee:

____________________________________
Wing-Ning Li

____________________________________
Brajendra Panda

____________________________________
Bill C. Hardgrave
©2007 by Joseph E. Hoag
All Rights Reserved
DISSERTATION DUPLICATION RELEASE

I hereby authorize the University of Arkansas Libraries to duplicate this dissertation when needed for research and/or scholarship.

Agreed ____________________________

Refused ____________________________
ACKNOWLEDGEMENTS

I thank my advisor, Dr. Craig Thompson, for his guidance and for his efforts to help me finish this dissertation. I also thank the remainder of my dissertation committee, Dr. Wing-Ning Li, Dr. Brajendra Panda and Dr. Bill Hardgrave, for their advice and guidance.

I also thank my lovely wife, Solveig, for the sacrifices that she made so that I could return to school and complete this degree.
# TABLE OF CONTENTS

1 Introduction ............................................................................................................. 1

1.1 Problem .................................................................................................................. 1

1.2 Thesis Statement ..................................................................................................... 2

1.3 Approach ................................................................................................................ 2

1.4 Potential Impact ..................................................................................................... 3

1.5 Related Work .......................................................................................................... 5

1.5.1 Commercial Data Generators ........................................................................... 5

1.5.2 Synthetic Data Generation Research ............................................................... 6

1.5.3 Data Inference and Privacy Issues .................................................................... 10

1.6 Organization of this Dissertation .......................................................................... 10

2 SDDL: A Language for Describing Synthetic Data Sets ........................................ 13

2.1 Database Element ................................................................................................. 16

2.2 Plug-in Functions .................................................................................................. 19

2.3 Pool Element ......................................................................................................... 22

2.4 Table Elements ..................................................................................................... 24

2.5 Field/Variable Elements ...................................................................................... 25

2.5.1 Formula Constraint ......................................................................................... 26

2.5.2 Min/Max Generation Constraint ...................................................................... 28

2.5.3 Dist (Distribution) Generation Constraint ...................................................... 28

2.5.4 QueryPool Constraint ...................................................................................... 30

2.5.5 Iteration Constraint .......................................................................................... 32

2.5.5.1 Query Iterations ........................................................................................... 33
2.5.5.2 Pool Iterations ................................................................. 35
2.5.5.3 Count Iterations ............................................................... 36
2.5.5.4 Cardinality in Iteration Elements ........................................ 37
2.5.5.5 Nesting Iterations ............................................................. 38
2.5.5.6 Iteration Variables ......................................................... 39

2.6 Summary .............................................................................. 41

3 SDG Software Architecture .................................................... 43

3.1 Top-Level Classes ............................................................... 44
3.1.1 Database Class ................................................................. 44
3.1.2 Tables and Fields .............................................................. 45
3.1.3 Pools and PoolChoices ...................................................... 48
3.1.4 Constants ......................................................................... 50

3.2 Generation Classes ............................................................. 51
3.2.1 Simple Generation Classes ................................................. 52
3.2.2 Complex Generation Classes .............................................. 55
3.2.2.1 Distributions and Tiers .................................................. 55
3.2.2.2 Formulas ....................................................................... 56
3.2.2.3 QueryPools .................................................................... 56
3.2.2.4 Iterations ....................................................................... 58
3.2.2.4.1 Count Iterations ....................................................... 63
3.2.2.4.2 Pool Iterations .......................................................... 64
3.2.2.4.3 Query Iterations ........................................................ 64

3.3 Parser-related Classes .......................................................... 65
5.6 A Word on Circular Data Dependencies .................................................. 144
5.7 Summary .................................................................................................. 147

6 Random but Deterministic ........................................................................ 149
6.1 Background .............................................................................................. 149
   6.1.1 Definitions of Terms ......................................................................... 149
   6.1.2 A Discussion of Streams .................................................................... 150
   6.1.3 Streaming Mechanisms ...................................................................... 153
       6.1.3.1 The Linear Streaming Mechanism (LSM) .............................. 153
       6.1.3.2 CMRG Streaming Mechanism (CSM) ................................. 155
   6.2 A Test for Randomness ........................................................................ 156
   6.3 Randomness Test Results ..................................................................... 158
       6.3.1 Producing the Random Number Streams .................................... 159
           6.3.1.1 RngInterface .................................................................... 159
           6.3.1.2 LSMRng .......................................................................... 160
           6.3.1.3 CSMRng .......................................................................... 160
           6.3.1.4 The Test Code .................................................................... 161
       6.3.2 Performance of Linear Streaming Mechanism .............................. 162
       6.3.3 Performance of CMRG Streaming Mechanism .............................. 164
   6.4 Summary ............................................................................................... 164

7 Generating Obfuscated Data ...................................................................... 166
7.1 A Spectrum for Data Obfuscation ............................................................ 166
   7.1.1 No Obfuscation (Level 0) ................................................................. 168
   7.1.2 Reversible Obfuscation (Levels 1-3) ................................................. 169
7.1.2.1 Partial Knowledge Reversible Obfuscation (Level 1) .................169
7.1.2.2 Process Reversible Obfuscation (Level 2).................................172
7.1.2.3 Combination Reversible Obfuscation (Level 3) .........................176
7.1.3 Irreversible Obfuscation (Levels 4-6)..........................................178
  7.1.3.1 Statistically Accurate or Dependency Preserving Irreversible Obfuscation (Level 4).................................................................................................179
  7.1.3.2 Domain Preserving Irreversible Obfuscation (Level 5)...............182
  7.1.3.3 Opaque Irreversible Obfuscation (Level 6)....................................184
7.2 Other Types of Obfuscation...............................................................185
  7.2.1 Schema Obfuscation ....................................................................186
    7.2.1.1 Attribute Suppression .................................................................186
    7.2.1.2 Generic Attribute Naming ............................................................186
    7.2.1.3 Attribute Decomposition .............................................................187
    7.2.1.4 Attribute Insertion ......................................................................188
  7.2.2 Cardinality Obfuscation ...............................................................188
  7.2.3 SDDL Support for Schema and Cardinality Obfuscation .............189
7.3 A Warning about Data Inference.......................................................191
7.4 Summary ............................................................................................192
8 Application: Generating Ten Years of Store/Item/Sales Data .............194
  8.1 Schema for Sales Table .................................................................194
  8.2 Generation Constraints ..................................................................195
  8.3 Enforcing Constraints with SDDL ..................................................197
    8.3.1 The item_dist Pool .................................................................197
8.3.2 The Sales Table ........................................................................................................199

8.4 Generation Results ....................................................................................................203

8.4.1 HP Enterprise Technology Center (ETC), Atlanta, GA ........................................203

8.4.2 HP Performance Center, Cupertino, CA ............................................................204

8.5 Summary ....................................................................................................................205

9 Application: Generating The Semantically Complex Hallux Application ......206

9.1 Description of Hallux Database ................................................................................206

9.2 Logical Challenges Encountered ............................................................................207

9.2.1 Order of Table Generation ....................................................................................207

9.2.2 Band/Album/Song Naming ....................................................................................210

9.2.3 Determinism Issues ...............................................................................................213

9.2.4 Differences in SQL Interpreters ............................................................................216

9.2.5 Using Temporary Attributes ................................................................................218

9.2.6 Ensuring Uniqueness in Key Values .....................................................................219

9.2.7 Using Scalable SQL Operations ............................................................................221

9.3 Generating the Data ..................................................................................................222

9.4 Summary ....................................................................................................................225

10 Application: Generating Legal Strings for Context-Free Languages ..........227

10.1 Definition of a Context-Free Grammar ....................................................................228

10.2 T*V* Form ...............................................................................................................228

10.3 Representing CFGs as Pools ....................................................................................230

10.4 SDDL Generation Code for CFGs ..........................................................................231

10.5 Example: Strings with an Even Number of 0s and 1s ............................................234
C.10.3 SDDL Description ............................................................................309
C.11 Contract .............................................................................................311
C.11.1 Schema ............................................................................................311
C.11.2 Rules .................................................................................................311
C.11.3 SDDL Description ............................................................................311
C.12 Band_Genre .......................................................................................312
C.12.1 Schema ............................................................................................313
C.12.2 Rules .................................................................................................313
C.12.3 SDDL Description ............................................................................313
C.13 Band_Member .....................................................................................314
C.13.1 Schema ............................................................................................314
C.13.2 Rules .................................................................................................314
C.13.3 SDDL Description ............................................................................314
C.14 Member_Instrument ..........................................................................315
C.14.1 Schema ............................................................................................315
C.14.2 Rules .................................................................................................315
C.14.3 SDDL Description ............................................................................315
C.15 Album ..................................................................................................316
C.15.1 Schema ............................................................................................316
C.15.2 Rules .................................................................................................317
C.15.3 SDDL Description ............................................................................317
C.16 Song ....................................................................................................318
C.16.1 Schema ............................................................................................319
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.22.1</td>
<td>Schema</td>
<td>331</td>
</tr>
<tr>
<td>C.22.2</td>
<td>Rules</td>
<td>331</td>
</tr>
<tr>
<td>C.22.3</td>
<td>SDDL Description</td>
<td>332</td>
</tr>
<tr>
<td>C.23</td>
<td>Performance</td>
<td>333</td>
</tr>
<tr>
<td>C.23.1</td>
<td>Schema</td>
<td>333</td>
</tr>
<tr>
<td>C.23.2</td>
<td>Rules</td>
<td>333</td>
</tr>
<tr>
<td>C.23.3</td>
<td>SDDL Description</td>
<td>333</td>
</tr>
<tr>
<td>C.24</td>
<td>Customer_Profile</td>
<td>334</td>
</tr>
<tr>
<td>C.24.1</td>
<td>Schema</td>
<td>334</td>
</tr>
<tr>
<td>C.24.2</td>
<td>Rules</td>
<td>335</td>
</tr>
<tr>
<td>C.24.3</td>
<td>SDDL Description</td>
<td>335</td>
</tr>
<tr>
<td>C.25</td>
<td>Customer_Genre</td>
<td>337</td>
</tr>
<tr>
<td>C.25.1</td>
<td>Schema</td>
<td>337</td>
</tr>
<tr>
<td>C.25.2</td>
<td>Rules</td>
<td>337</td>
</tr>
<tr>
<td>C.25.3</td>
<td>SDDL Description</td>
<td>337</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2-4: SDDL Element Hierarchy ................................................................. 15
Figure 2-5: Sample Directed Graph ................................................................. 39
Figure 3-1: Database Class ........................................................................... 44
Figure 3-2: Table and Field Classes ............................................................... 47
Figure 3-3: Pool and PoolChoice Classes ....................................................... 49
Figure 3-4: Constant Class ......................................................................... 50
Figure 3-5: Simple Generation Classes ......................................................... 52
Figure 3-6: Distribution, Formula and QueryPool Classes ............................ 58
Figure 3-7: Iteration Classes ....................................................................... 60
Figure 3-8: Iteration.generateValue() Logic ............................................... 62
Figure 3-9: Iteration.bump() Logic ................................................................. 63
Figure 3-10: Parser-related Classes ............................................................... 66
Figure 3-11: Parse Tree for Formula $F_1 \times 10 + 4$ ..................................... 68
Figure 3-12: Parse Tree for Query "SELECT name, salary FROM employees
WHERE age = [F2] ORDER BY salary" ...................................................... 69
Figure 3-13: Parse Tree for Formula $0 + \text{states[state].cities["New York"]}.\text{population}$ ................................................................. 70
Figure 3-14: Parse tree for formula $\text{states["New"}+" +" \text{York"]}+5.\text{cities}$ ... 71
Figure 3-15: Random Number Generation Classes ....................................... 73
Figure 3-16: Environment Classes ................................................................. 74
Figure 3-17: PluginFunctionInterface ............................................................ 75
Figure 3-18: SaxDBUnmarshaller Class .......................................................... 77
Figure 3-19: Globals Class .................................................................78
Figure 3-20: Main Class .................................................................80
Figure 3-21: Program Execution Sequence Diagram ...................................83
Figure 3-22: Sequence Diagram for Parsing a Pool .......................................84
Figure 3-23: Sequence Diagram for Parsing a Table .......................................86
Figure 3-24: Sequence Diagram for Parsing a Formula .....................................89
Figure 3-25: Sequence Diagram for Generating Data ......................................90
Figure 4-1: Parallel Data Generation ........................................................93
Figure 4-2: Map modeled by "map" Pool ................................................105
Figure 5-1: E-R Diagram for "Sells In" Relation ..........................................121
Figure 5-2: Weak Entity Example ..........................................................143
Figure 5-3: Weak Entity Specified Through Cardinality Constraints ...............143
Figure 5-4: Circular Dependencies in TPC-H Tables ....................................145
Figure 6-1: Badly chosen stream starting points .........................................152
Figure 6-2: Optimally chosen starting points for 4 streams ..........................153
Figure 6-3: Hyperplaning in 2-dimensional space .......................................155
Figure 9-1: Hallux Database E-R Diagram ..............................................208
Figure 10-1: "Even Number of Zeroes and Even Number of Ones" DFA ..........234
1 INTRODUCTION

1.1 Problem

There is an increasing need for synthetic data in the IT industry. Applications that benefit from synthetic data include (but are not limited to):

- **Regression testing.** Repeatedly generate the same large data set for testing enterprise applications. A synthetic data generator allows one to “grow” exactly the same large data set repeatedly when needed, rather than storing it between tests. This is especially important for data sets in the giga- or terabyte range.

- **Secure application development.** Develop enterprise applications using generated data that is realistic but not real. A synthetic data generator can produce a “sanitized” copy of a table, which retains many of the original table’s data characteristics but does not contain personally identifiable information. Third parties can develop software and test it on realistic data sets without exposing real data.

- **Try-before-you-buy.** A vendor wants to demonstrate an enterprise application to a potential customer. For instance, Oracle wants to demonstrate Oracle 10g (grid-based Oracle) to a customer who has not yet moved to a grid. The customer does not want to commit to buying a grid and 10g until they can see how their applications would run in that environment.
• **Testing of data mining applications.** Generate data sets with known characteristics to gauge whether or not data mining tools can discover those characteristics.

Of course, one can always use a general-purpose programming language like C++ or Java as a generator to construct a special-purpose data set or alternately can use SQL to extract scrubbed subsets from a database management system. But, can a general-purpose description language be developed that contains powerful description and constraint mechanisms? And can a generation engine be developed that could read descriptions in such a language and produce data sets from them? Finally, can this generation engine produce data in a timely fashion?

1.2 **Thesis Statement**

A synthetic data generator can be constructed to represent and generate a wide variety of simple-to-complex structured data sets including not only traditional tabular but also grammar and graph-based data sets. Data generation constraints can be captured in a well-defined description language. The synthetic data generator can be designed to execute efficiently and run in parallel to generate very large data sets. Synthetic data generation has theoretical underpinnings in the areas of random number theory, database representational concepts and data obfuscation.

1.3 **Approach**

An overview of the research methodology follows:

• Identify related work in synthetic data generation along with its limitations.
• Develop a language for describing and constraining synthetic data. This language will be called Synthetic Data Description Language (SDDL).

• Develop an engine that reads SDDL files as input and produces the described data sets as output. This engine needs to be able to run in parallel, to take advantage of the parallelism available in modern cluster computing technology and produce data sets in a timely fashion. Also, the engine needs to be deterministic in that it will produce the same output (given the same input) every time regardless of the degree of parallelism employed during generation.

• Investigate theoretical foundations of synthetic data generation.

• Show the flexibility and power of the synthetic data generation framework by applying with clear benefit to several application problems.

1.4 Potential Impact

High-level programming languages like C/C++, C# and Java provide useful abstractions for software developers, and result in a significant time savings over programming in assembly language. Likewise, a synthetic data description language could provide nice abstractions for those responsible for generating synthetic data sets, saving them the time associated with writing an application-specific data generator “from scratch.” To achieve maximal impact, a synthetic data generation framework should have the following properties:

• **Descriptiveness and Realism.** The description language should provide mechanisms sufficient to model a wide range of data sets. Also, a
description language should allow for various levels of obfuscation of an existing table. The degree of descriptiveness associated with the description language directly affects the degree of realism possible in the generated data.

- **Determinism.** The data set generated from an input file should be the same from run to run, regardless of the platform or the degree of parallelism used during data generation. This property is necessary for supporting regression testing applications.

- **Speed.** Speed is always associated with convenience. To achieve maximum speed, a generation engine should support parallel data generation.

- **Extensibility.** The user should be able to extend existing built-in or pre-existing libraries with new data dictionaries, operations, and table descriptions.

- **Usability.** The user should understand how to use the constructs of the generator to specify potentially complex, large datasets faster using the generator than if they accomplish the same thing with some other manual method.

Research on large-scale scientific, military and business problems often begins with assembling custom data sets that can be used to test the developing applications. This has been true in our own work. Over the past two years (as will be described in later chapters), our evolving synthetic data generation capability has benefited several diverse research projects in applications including customer data integration, data indexing, RFID
scalability testing, supply chain simulations, product try-before-you-buy testing, and other areas. A software architect in the Information Technology division of a Fortune 100 company reviewed our work on a synthetic data generation framework and stated “this could save our company a million dollars a year.” The need for sophisticated synthetic data generation tools definitely exists, and this research will take a step toward filling that need.

1.5 Related Work

1.5.1 Commercial Data Generators

Synthetic data generation is an emerging technology. Commercial synthetic data generation products have recently become available [1,2,3,4]. These products do a good job at producing moderate amounts of simply defined data. Their user interfaces are polished and intuitive; they usually come with useful “canned” data dictionaries; and they are probably the best option for small, simple jobs. These products have limited range of representation and there are some types of relations, functional dependencies, and intra- and inter-table constraints that are not easy to describe using these commercial products. This class of generation products runs on a single machine, which puts a ceiling on their rate of data generation. In order to generate “industrial strength” (very large and complex) data sets in a timely fashion, a generator needs to be able to execute in parallel across a number of fast processors with an expressive and extensible representational coverage.
1.5.2 Synthetic Data Generation Research

In 1994, Gray et al. [5] described techniques for generating special-purpose data sets in parallel. The paper described the generation of the “Accounts” table for the TPC-A benchmark [6]. This paper also described some useful parallel techniques for generating dense-unique-pseudo-random sequences (sequences of $N$ numbers in which all numbers $1..N$ are represented, each number appears exactly once, and the sequence appears to be in random order), as well as non-dense, non-uniform distributions (examples of non-uniform distributions include Gaussian, Zipfian and Poisson distributions).

In 2000, a group from Polytechnic University, Lucent Technologies and AT&T Labs published a paper [7] describing a “tool for populating the database with meaningful data that satisfy database constraints”. This tool accepted text files as input that specified possible discrete values (or value sets) for individual attributes in a table, and also had the capability of weighting values (or value sets). The format of these input text files could be considered a simplistic precursor to a synthetic data description language\(^1\).

In 2001, a team of researchers from the University of Wisconsin published a paper [8] that described a “data generator for generating synthetic complex-structured XML data that allows for a high level of control over the characteristics of the generated data”. It goes on, “The goal of the data generator is rather to generate XML data sets with widely varying characteristics by varying the input parameters, thereby covering many different parts of the space of possible XML data sets. This allows the data

\(^1\) Input language examples from other synthetic data generation frameworks will be provided in Chapter 2.
generator to be used in a wide range of applications to gain insights into the performance of proposed techniques on different kinds of XML data.” This effort is a good example of a generator that produces output that is not strictly tabular.

In 2004, Stephens and Poess published a paper on MUDD, a multi-dimensional data generator [9]. MUDD was built with the purpose of generating data for the TPC-DS benchmark, but could be extended to other applications of synthetic data generation. MUDD shows impressive generation speed and scales linearly when exploiting multiple processors. The MUDD input description language is an improvement on [7] in that it is capable of multi-field, multi-weight step distributions.

In 2004 Jonathan White, a graduate student at the University of Arkansas, developed a special-purpose synthetic data generator [10]. White’s data generator was limited to 50 or so field types associated with a mailing list application, such as name, address, city, state, zip code, area code, phone number, etc… This generator produced very realistic data for the field types that it supported, but was not extensible.

In 2005-2006 a team from the University of California at Riverside and Lucent Technologies developed a synthetic data generator called IDSG [11, 12]. IDSG stands for “IDAS Data Set Generator”, and IDAS stands for “Information Discovery and Analysis Systems”. The goal of IDSG is to produce test data sets for various IDAS. IDSG excels at preserving multi-dimensional dependencies between data attributes. IDSG provides a web interface through which clients can select a data set (i.e., “Credit Card Data”, “Names”) to generate; whereupon the resulting comma-separated value (.csv) files are compressed and made available to the client. To the author’s knowledge, however, IDSG provides no easy way to describe and generate user-specified data sets.
In 2005 researchers at Microsoft described the most sophisticated language approach to that point for synthetic data generation [13]. Their description language (“Data Generation Language”, or DGL) is a C-like language that contains a set of mechanisms primarily used for specifying synthetic data distributions. There is no indication in [13], however, that any attempt was made to use DGL to generate data in parallel.

A team from Aalborg University reported [14] in 2006 on a synthetic data generator that they later christened “KRDataGeneration” and made into a commercial product [15]. This generator presents a graph-model approach for describing synthetic data. This graph-model approach excels at handling large schemas and complicated dependencies. However, the graph-model approach allows for only coarse control in specifying the manner in which data should be generated; a language approach allows for much finer control on data description and constraints. KRDataGeneration also claims the ability to generate data in a distributed fashion to improve generation speed, but no details regarding the distributed generation logic are provided.

Taken as a whole, the SDG/SDDL framework compares favorably with previous generation frameworks by fulfilling the requirements mentioned in Section 1.4:

- **Descriptiveness and Realism.** The pool and iteration variable constructs found in SDDL allow for the modeling of very complicated concepts. In addition to tabular data, SDDL can model and generate data for such things as graphs, maps and grammars. And, while SDDL may not explicitly support some of the more complicated distributions (such as Zipfian and Poisson distributions) supported in [13], these distributions
can be enforced through the use of user-defined plug-in functions. This
descriptiveness allows for very realistic data to be generated.

- **Determinism.** The output of the generation engine, given a constant input
  file, will be the same from run to run regardless of the degree of
  parallelism or the platform used during data generation.

- **Speed.** SDDL was designed to be run in parallel, which allows for fast
data generation. The SDG generation engine contains automatic
  partitioning logic, which means that generation distribution details need
  not be included in the data description (also claimed by [9]).

- **Extensibility.** SDDL is easily extensible. Users can define their own data
  dictionaries (in the form of pools), their own operations (in the form of
  plug-in functions), and their own synthetic data descriptions (in the form
  of table definitions).

- **Usability.** Another graduate student at University of Arkansas, Josh Eno,
developed a graphical user interface for SDG. One of the nice features of
  this interface is that it can connect to one or more remote relational
  databases and import metadata including schema information. However,
  this interface is at present less mature than the rest of SDG and so in this
  one area, SDG currently lacks the full expressive usability that the SDDL
  language supports, and so some commercial data generation products have
  an advantage.
1.5.3 Data Inference and Privacy Issues

The inference problem occurs when data values considered non-sensitive can be combined together (or with external knowledge) to infer data values considered sensitive. The data inference problem has been chronicled in several articles, including [16, 17, 18, 19].

One possible application for a general-purpose synthetic data generator is to extract data from a sensitive table and produce a sanitized table suitable for public access. In [17], Bakken et al provide a taxonomy of the different methods typically employed to effect data obfuscation. Portions of the taxonomy described in [17] will be used in Chapter 7 to describe how SDG can be used to provide obfuscated versions of sensitive data.

1.6 Organization of this Dissertation

Chapters 2-4 describe the Synthetic Data Generation (SDG) framework and Synthetic Data Description Language (SDDL):

- Chapter 2 contains a detailed description of SDDL. As the strength of the generator lies in the power of the input language, SDDL is the logical place to begin describing the generator. This chapter enumerates all of the description and constraint constructs provided by SDDL, and give examples of each.

- Chapter 3 contains architectural design details for the data generation engine, including class diagrams and sequence diagrams.
• Chapter 4 contains a description of the partitioning algorithms employed by SDG that allow for parallel data generation. This chapter also explores the use of iteration variables to effect tight inter-row dependencies without compromising parallel data generation ability or efficiency.

Chapters 5-7 explore theoretical underpinnings of synthetic data generation:

• Chapter 5 answers some questions about the representational adequacy of SDDL. SDDL will be measured as to how well it supports various descriptional concepts found in the relational model and E-R diagrams. Examples of such descriptional concepts include domain constraints, referential integrity constraints, and cardinality constraints.

• Chapter 6 examines random number theory, and how the SDG framework employs it. The SDG generation engine must produce random but deterministic data, which can be tricky to implement. There are good ways and bad ways to implement “random but deterministic”, and this chapter will explain how SDG avoids the pitfalls.

• Chapter 7 explores the concept of data obfuscation, which is a necessary security and privacy measure employed when exporting data to a third party. A spectrum of data obfuscation techniques are presented, and other types of obfuscation are presented as well. All obfuscation techniques are accompanied by SDDL examples enforcing that obfuscation technique.
Chapters 8-11 describe disparate applications of synthetic data generation and what we learned from each:

- Chapter 8 describes the synthetic generation of ten years of store-item-sales data for a major retailer. This application demonstrated the generation speed made possible by the parallelization of the generation process. Terabyte-sized data sets resulted from this effort.

- Chapter 9 describes the generation of a fictitious but realistically complex music industry database. This application tackled many of the challenges associated with typical industry databases, including multiple inter-related tables and the incorporation of business rules into the generation description.

- Chapter 10 explores the generation of legal strings for user-specified context-free languages. This application is an example of a non-tabular application, and could be the basis for many more useful applications.

- Chapter 11 describes the use of SDG and SDDL to produce parallel mailing lists, one of which is a “canonical” fixed-format list, the other of which is a free-format list seeded with errors. This is another non-tabular application, and it has real-world importance.

Chapter 12 gives some conclusions and describes areas for potential future work.
2 SDDL: A LANGUAGE FOR DESCRIBING SYNTHETIC DATA SETS

Some of the previous synthetic data generation frameworks developed some sort of input language for describing the characteristics of data to be generated. The language presented by Chays et al. [7] allowed for the enumeration of discrete choices (or choice groups) for each field, and allowed choices (or choice groups) to be weighted. Figure 2-1 shows a sample column description from [7]; in this city column, 90% of the values will be “domestic” (one of Brooklyn, Florham-Park, or Middletown), and 10% of the values will be “foreign” (one of Athens or Bombay).

```plaintext
city:
  -choice_name: domestic
  -choice_prob: 90
Brooklyn
Florham-Park
Middletown
--
  -choice_name: foreign
  -choice_prob: 10
Athens
Bombay
```

Figure 2-1: Sample Input File from [14]

The language associated with the MUDD generator [9] was similar to [7]; it allowed for weighted discrete field values. The MUDD language was an improvement on [7] in that it allowed for multiple fields and multiple weights per step. Figure 2-2 shows a sample data description from [9]. Each step contains two data columns (day-of-year and date) and two weight columns (one for a non-leap year, one for a leap year). Thus, during a non-leap year the sales are governed by the left-hand weights, and during a leap year the sales are governed by the right-hand weights.
The most sophisticated attempt at a data description language was developed by Bruno and Chaudhuri [13] with their Data Generation Language (DGL). DGL is a C-like language that allowed users to specify complicated data distributions. DGL also allowed for querying existing data during the generation of new data. Figure 2-3 shows a DGL example taken from [13]. Apparently, it results in a 10,000 2D points being generated, where 65% of the rows being uniformly distributed between (5,7) and (15,13), and 35% of the rows being normally distributed with a mean of (5,5) and a standard deviation of (1,2).

```
LET count=1000, 
P=0.65, 
U = Uniform([5,7],[15,13]), 
N = Normal([5,5],[1,2]), 
IN Top(ProbUnion(U, N, P), count)
```

Figure 2-3: Sample DGL

In our work, the Synthetic Data Description Language (SDDL) was developed as the companion language for SDG. SDDL is a hierarchical XML-based language designed to describe synthetic data sets. SDDL provides mechanisms for defining user-
specified domains and crafting detailed models of the real world, and allows for the
description of complex intra-row, inter-row and inter-table dependencies. It combines
many of the features of [7], [9] and [13], while also introducing some new functionality.

This chapter provides a detailed description of SDDL, followed by a brief
summary of how SDDL compares to these other description languages.

Before going into detail, a brief overview is in order. SDDL elements are
organized hierarchically as follows:

![SDDL Element Hierarchy Diagram]

Figure 2-4: SDDL Element Hierarchy
Individual SDDL elements will be explained in more detail in succeeding sections. Each section will begin with a formal DTD specification of its SDDL element(s), and will then go into a description of the elements. (See Appendix A for a full DTD for SDDL).

2.1 Database Element

DTD:

```xml
<!ELEMENT database (seed? fieldSep? import* load* constant* pool* table+)>
<!ELEMENT seed (#CDATA)>
<!ELEMENT fieldSep (#CDATA)>
<!ELEMENT import EMPTY>
<!ATTLIST import filename CDATA #REQUIRED>
<!ELEMENT load EMPTY>
<!ATTLIST load funcName CDATA #REQUIRED>
<!ELEMENT constant EMPTY>
<!ATTLIST constant name CDATA #REQUIRED>
<!ATTLIST constant type CDATA #REQUIRED>
<!ATTLIST constant value CDATA #REQUIRED>
```

The database element is the outer-most element in an SDDL file; it contains all information necessary to generate a synthetic data set. A database element can contain:

- An optional `seed` element. The user can specify the initial seed used for random number generation; this defaults to 0. By changing the initial seed, the user can change the data sets output by the generator.

- An optional `fieldSep` element. The user can define the field separator character; this defaults to a comma (,).
• 0 or more import elements. An import element (<import filename="foo"/>) allows for modular SDDL data set definition. If a certain pool is used by multiple SDDL files, it is efficient to import the pool rather than to copy it to multiple SDDL files.

• 0 or more load elements. A load element (<load funcName="fxn"/>) allows for the use of user-specified plugin functions.

• 0 or more constant elements. Constants are a convenience mechanism for parameterizing complex description files.

• 0 or more pool elements. Pool elements allow the user to define domains or models for use during generation.

• 1 or more table elements. Tables allow the user to explicitly describe the generation rules for the tables being generated.

A simple example containing most of these elements is in order. Suppose there is a file called colors.sddl that contains the following:

```xml
<pool name="colors">
  <choice name="red"/>
  <choice name="green"/>
  <choice name="blue"/>
</pool>
```

Now suppose that the main SDDL file looked like this (line numbers added for explanatory purposes):

1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <database>
3.  <seed>1240958412</seed>

17
Line 2 begins the database element. Line 3 specifies a beginning random seed value. Line 4 specifies that the field separator should be a vertical bar (|), rather than the default comma (,) value. Lines 5 and 6 declare constant values slope and intercept. Line 7 imports the colors pool from the file colors.sddl.

Lines 8-19 specify a table named POINTS. POINTS has 3 attributes: X, Y, and color. Each X,Y pair produced will represent a point on the line Y=slope*X+intercept, where slope and intercept are defined as 2 and 10, respectively, in lines 5 and 6. The color field will be one of red, green or blue.

The data file generated from the SDDL file above looked like this:

45.28|100.56|'red'
39.18|88.36|'blue'
89.66|189.32|'green'
81.85000000000001|173.70000000000002|'red'
14.97|39.94|'blue'
2.44|14.879999999999999|'blue'
22.04|54.08|'red'
40.6|91.2|'green'
2.2 Plug-in Functions

SDDL and SDG support the loading of user-defined plug-in functions at generation time. Plug-in functions must be constructed in a precise manner in order to be successfully loaded by SDG.

All plug-in functions are really classes that implement the following interface:

```java
public interface PluginFunctionInterface {
    public Lexeme call(ArrayList<Lexeme> params,
                        Environment env,
                        RngInterface rnd)
                        throws sdg.EvaluationException;
}
```

The name of the class is always the name of the plug-in function, and the implementation of the function is always done via the `call()` method.

Suppose that one wanted to create a plug-in function named `uppercase` that accepts a single string parameter and outputs the input string with all characters converted to upper case. One would create a file called `uppercase.java`, and then populate that file as follows:

```java
import java.util.ArrayList;
import sdg.*;

/*
 * Implementation of plugin function uppercase().
 *
 * Usage: uppercase(String input)
 *
 * Returns the input string with all characters converted to upper case.
 *
 * Returns a STRING Lexeme.
 */
```
public class uppercase implements PluginFunctionInterface {
    public Lexeme call(ArrayList<Lexeme> params, Environment env, RngInterface rnd) throws EvaluationException {
        // Check for the presence a single parameter.
        if((params == null) || params.size() != 1) {
            throw new EvaluationException("uppercase(): Expected 1 parameter.");
        }

        // Evaluate all parameters
        Lexeme eOrig = params.get(0).evaluate(env);

        // Check that parameter is of type STRING
        if(eOrig.getType() != Lexeme.STRING) {
            throw new EvaluationException("uppercase(): Arg 1 (input) needs to be a STRING");
        }

        // Extract input string from Lexeme
        String orig = eOrig.getStringValue();

        // Convert string to upper case
        String converted = orig.toUpperCase();

        // Construct a lexeme, fill it with our value, // and return.
        Lexeme rval = new Lexeme();
        rval.setValue(converted);
        rval.setType(Lexeme.STRING);
        return rval;
    }
}

Having constructed the plug-in function class, one would then need to compile it.

On the compilation line, the file sdg.jar would need to be included in the compilation classpath. The compilation line would look something like:
javac -classpath ..\lib\sdg.jar uppercase.java
The result is a class file, uppercase.class. That class file can now be used as a plug-in function in an SDDL file. Here is an example of the use of the uppercase() plug-in function within an SDDL file:

```xml
<database>
  <seed>87367811</seed>
  <load funcName="uppercase"/>
  <pool name="colors">
    <choice name="red"/>
    <choice name="green"/>
    <choice name="blue"/>
  </pool>
  <table name="scratch">
    <field name="F1" type="CHAR(5)">
      <iteration pool="colors"/>
    </field>
    <field name="F2" type="CHAR(5)">
      <formula>uppercase(F1)</formula>
    </field>
  </table>
</database>
```

Note that only the name of the function (uppercase) is used as the value for the funcName attribute, NOT the name of the class file (uppercase.class). SDG will automatically look for uppercase.class if given uppercase as the function name.

The SDDL above defines a table with two fields: F1 iterates through the choices of the colors pool, and F2 is an upper-case version of F1. The table produced looks like this:

<table>
<thead>
<tr>
<th>F1</th>
<th>F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>RED</td>
</tr>
<tr>
<td>green</td>
<td>GREEN</td>
</tr>
<tr>
<td>blue</td>
<td>BLUE</td>
</tr>
</tbody>
</table>

Table 2-1: Table Showing Successful Results of Plug-in Function Use
2.3 Pool Element

DTD:

```xml
<!ELEMENT pool (choice+)>
<!ATTLIST pool name CDATA #REQUIRED>

<!ELEMENT choice (pool | ANY)>
<!ATTLIST choice name CDATA #REQUIRED>
```

Pools are used to describe domains or models to be used during data generation. A pool element consists of one or more choice elements. A choice element can be empty, but can also contain sub-pools and auxiliary data elements. Each pool and choice element must have a name associated with it.

A simple pool might look like this:

```xml
<pool name="states">
  <choice name="AR"/>
  <choice name="AZ"/>
</pool>
```

Pool elements are accessed via pool references, which are part of the formula grammar for SDDL. The following field definition would access the pool above:

```xml
<field name="state" type="CHAR(2)">
  <formula>states</formula>
</field>
```

The field above would generate values that were either AR or AZ.

However, a pool can also be a more complicated, hierarchical structure like this:

```xml
<pool name="states">
  <choice name="AR">
    <fullName>Arkansas</fullName>
    <weight>60</weight>
    <pool name="universities">
      <choice name="University of Arkansas"/>
    </pool>
  </choice>
</pool>
```
<choice name="Harding University"/>
<choice name="John Brown University"/>
</pool>
</choice>
<choice name="AZ">
<fullName >Arizona</fullName >
<weight>40</weight>
<pool name="universities">
<choice name="Arizona State University"/>
<choice name="University of Arizona"/>
<choice name="Northern Arizona University"/>
</pool>
</choice>
</pool>

Note the following about the states pool above:

- Pools can be nested within pools. Each choice in the top-level states pool has a universities pool. This nesting can be arbitrarily deep.

- Elements can be weighted or unweighted. The choices in the top-level states pool have weights associated with them; therefore, there will be a 60% chance of picking AR and a 40% chance of picking AZ. The choices in the nested universities pools are unweighted; all entries in these pools have an equal chance of being selected.

- Elements can contain arbitrary attributes. Each element in the top-level states has an attribute called fullName.

The following SDDL field definitions show how to reference the more complicated pool above:

```
<field name="state" type="CHAR(2)">
  <formula>states</formula>
</field>
<field name="stateName" type="CHAR(16)">
  <formula>states[state].fullName</formula>
</field>
```
The state field above will generate the value AR 60% of the time, and the value AZ 40% of the time, based on the weights specified in the states pool. The stateName field above would generate the full name of the state generated by the state field. The university field above would generate one of the values in the university sub-pool of the chosen state value.

2.4 Table Elements

DTD:

```xml
<!ELEMENT table ((field | variable)+ sql*)>
<!ATTLIST table name CDATA #REQUIRED>
<!ATTLIST table length CDATA>
<!ATTLIST table fileSuffix CDATA>
<!ELEMENT sql (#CDATA)>
```

A table element is the outer element in a table description. A table element must have an accompanying name attribute, and can optionally have length and fileSuffix attributes. A length attribute is necessary for specifying the number of rows in tables that do not contain iterations, and is ignored for tables that do contain iterations. The fileSuffix attribute can be used to specify a file suffix for tables that are generated to text files; it defaults to .csv. A table element contains one or more field/variable elements.

A table element may also contain optional sql elements. An sql element is an arbitrary SQL directive that is processed after the table is created. An sql element is
only used when generating data directly to a database, and allows for such actions as the creation of indices, the removal of attributes that are not longer needed, and the creation of views. Consider the following example:

```xml
<table name="foobar">
  <field name="F1" type="int">
    [...]
  </field>
  [...]
  <sql> create index F1_index on foobar(F1)</sql>
</table>
```

In the example above, an index named F1_index would be created on the F1 field of the foobar table just after the foobar table was generated.

### 2.5 Field/Variable Elements

**DTD:**

```xml
<!ELEMENT field (formula|(min max step?)
  |dist|iteration|queryPool)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field type CDATA #REQUIRED>
```

The field element is the building block of the table element. A field element describes the name, domain type, and generation constraints associated with a column of the table. A field element must have accompanying name and type attributes, and contains a single generation constraint element.

Field type values can be one of int, real, string, boolean, date, time, or timestamp. Field type values can also be expressing using equivalent SQL types ("CHAR(64)", "INTEGER", "DECIMAL(7,2)", etc...).
A variable element is nearly identical to a field element: they are both building blocks for table elements, and they are both defined identically. The only difference is that a field element will result in an output column, and a variable element will not. Variable elements are used as "placeholders" in complicated equations.

There are five types of generation constraint element types: min/max, distribution, queryPool, iteration and formula. Each will be described in turn in the subsections below. By default, if no generation constraint type is specified for a field, a min/max constraint will be used for the field.

2.5.1 Formula Constraint

DTD:

```xml
<!ELEMENT formula (#CDATA)>
```

Because formula expressions can be used as the basis for defining other types of constraints, like min/max and statistical distribution constraints, they are mentioned first. A formula constraint is the most powerful of all constraints. A formula is an arbitrary expression. The expression can contain numerical/string/time constants, SDDL constants, values of previously generated fields (or variables) from the current table, iteration variables, pool references, built-in functions, and a variety of operators. Some examples of formulas will be presented in this subsection; see Appendix B for a complete description of formula expression grammar and semantics.

Consider the following example:

```xml
<pool name="Degrees"/>
```
The Degrees pool contains a number of degree types and a hypothetical salary range for each. The Employees table generated contains 3 attributes. The EmpID field iterates from 100 through 109. The Degree field is constrained by a formula containing a simple pool reference; the value for Degree will be one of None, BS, MS and PhD. The Salary field is constrained by a min/max constraint with formula expressions for values; the value for Salary will be in the range indicated by the chosen degree.
2.5.2 Min/Max Generation Constraint

DTD:

```xml
<!ELEMENT min (#CDATA)>
<!ELEMENT max (#CDATA)>
<!ELEMENT step (#CDATA)>
```

A min/max constraint is the simplest of all forms of data constraints. It simply specifies a range of values for the field, and results in data uniformly distributed over that range. Consider the following field descriptions:

```xml
<field name="salary" type="int">
  <min>20000</min>
  <max>95000</max>
</field>

<field name="hiredate" type="date">
  <min>#1980-01-01#</min>
  <max>#2006-10-31#</max>
</field>
```

The values for the salary column would be random integers in the range 20000–95000 (inclusive). The values for the hiredate column would be random dates between 1980-01-01 and 2006-10-31 (inclusive).

Note that values for min and max can also be specified using formula expressions. See the example given for this previously in section 2.5.1.

2.5.3 Dist (Distribution) Generation Constraint

DTD:

```xml
<!ELEMENT dist (tier+)>
<!ELEMENT tier EMPTY>
```
A distribution generation constraint allows for more statistical control than a simple min/max constraint. A dist element is composed of one or more tier elements. Each tier element must have prob, min, and max attributes. The probabilities of all tiers in a distribution must add to 1.

The salary and hiredate fields from section 2.5.2 could have been specified as follows:

```xml
<field name="salary" type="int">
  <dist>
    <tier prob="0.60" min="20000" max="35000"/>
    <tier prob="0.30" min="35001" max="50000"/>
    <tier prob="0.10" min="50001" max="90000"/>
  </dist>
</field>

<field name="hiredate" type="date">
  <dist>
    <tier prob="0.90" min="#2004-01-01#" max="#2006-10-31#"/>
    <tier prob="0.10" min="#1980-01-01#" max="#2003-12-31#"/>
  </dist>
</field>
```

The salary column above would now have 60% of its values in the 20000–35000 range, 30% of its values in the 35001–50000 range, and 10% of its values in the 50001–90000 range. The hiredate column (perhaps reflecting a high turnover rate) would show that 90% of employees hired on in the last 3 years or so, while only 10% of employees have been with the company since before 2004.

Note that the prob, min and max attributes for a tier element can also be described using a formula expression:
In the SDDL definition for the `hiredate` field above, 90% of the `hiredate` values will be somewhere in the last 2 years, and 10% will be somewhere between 2 and 20 years ago.

### 2.5.4 QueryPool Constraint

**DTD:**

```xml
<!ELEMENT queryPool EMPTY>
<!ATTLIST queryPool query CDATA #REQUIRED>
```

A `queryPool` constraint forces the column values to be chosen from the results of a query. A `queryPool` element contains no other elements, and requires a `query` attribute providing the associated query. The `query` attribute is an SQL string that may be parameterized. A `queryPool` constraint is a good way to enforce referential integrity between your generated table and a pre-existing table.

Consider the following SDDL snippet:

```xml
<field name="StudentID" type="int">
  <queryPool query="SELECT ID FROM students"/>
</field>
```
In this example, the StudentID column values are guaranteed to be legal IDs from the students table, thereby assuring referential integrity between the table being generated and the students table.

Note that there is no guaranteed order to the column values; the ID values from the students table are chosen in random order, and they may be repeated. If you would like to output the values of a query in order, consider using the query iteration mentioned in section 2.5.5.1.

QueryPools also support parameterized queries, and can retrieve multiple columns, as demonstrated by the following field definitions:

```xml
<field name="ageField" type="int">
  <min>20</min>
  <max>65</max>
</field>
<field name="LastName" type="CHAR(32)">
  <queryPool query="SELECT LastName, FirstName FROM employees WHERE age = [ageField]"/>
</field>
<field name="FirstName" type="CHAR(32)">
  <formula>LastName$FirstName</formula>
</field>
```

The LastName field contains a queryPool, and will return the left-most column value (in this case, LastName) from the query result. Other column values can...
be accessed by using the sub-field ($) operator on the LastName field, as shown in the FirstName field formula.

The query string format can be defined in pseudo-BNF as follows:

\[
\text{STRING \ [OBRACKET VARIABLE CBRACKET} \ [STRING] \ ]^*
\]

The following would all be valid query strings:

“SELECT name, age FROM employees”

“SELECT name, age FROM employees WHERE salary < 50000”

“SELECT name, age FROM employees WHERE salary < [salary]”

“SELECT name, age FROM employees WHERE salary < [salary] and gender = [gender]”

The first two query strings above are not parameterized. The third query string has a single parameter (the value of a field/variable named salary) and the fourth query string has two parameters (the values of fields/variables named salary and gender).

If a query results in a NULL data set being returned, then QueryPool will generate the value “-1” for string fields, -1 for integer fields, and -1.0 for real fields. Return results for other types of fields, in the case of a NULL data set, are undefined and may cause the SDG generation engine to throw an exception.

2.5.5 Iteration Constraint

DTD:

```xml
<!ELEMENT iteration (((repeatMin repeatMax) | repeatDist)? itervar*)>
<!ATTLIST iteration base CDATA>
<!ATTLIST iteration count CDATA>
<!ATTLIST iteration query CDATA>
<!ATTLIST iteration pool CDATA>
```
An iteration constraint causes field values to iterate through a specified set of values. That set of values may be associated with a query, a pool, or a set of integers. Iteration elements can be logically nested (though you cannot nest iteration elements in the XML sense). If fields with iteration constraints are present in a table element, then the length of the associated table will be determined by the result of the query.

2.5.5.1 Query Iterations

A query iteration causes the values of its associated column to iterate through the results of a query. A query iteration is called for by including a query attribute with the iteration element. Consider the following:

```xml
<field name="ID" type="int">
  <iteration query="SELECT ID FROM students"/>
</field>
```

In this example, the ID column value would iterate through all values of ID in the students table.

It is also possible to use query iterations for "pass-through" functionality, using the subfield ($) operator supported by formula expressions. For example, suppose
you have a source table called `sensitive` that has ID, name, SSN, Degree, and Salary attributes. You could completely copy this table using the following SDDL:

```
<table name="sensitive-copy">
  <variable name="query" type="string">
    <iteration query="SELECT ID,Name,SSN,Degree,Salary FROM sensitive"/>
  </variable>
  <field name="ID" type="int">
    <formula>query$ID</formula>
  </field>
  <field name="Name" type="string">
    <formula>query$Name</formula>
  </field>
  <field name="SSN" type="CHAR(11)">
    <formula>query$SSN</formula>
  </field>
  <field name="Degree" type="CHAR(3)">
    <formula>query$Degree</formula>
  </field>
  <field name="Salary" type="int">
    <formula>query$Salary</formula>
  </field>
</table>
```

The `query` variable reads in all rows of the query and iterates through them. The remaining fields access specific columns of the query results.

Of course, copying tables is not all that impressive. Query iterations (and the pass-through operator ($)) can be used to provide partial tables, effectively creating a sanitized copy of the source table. Some fields can be "passed through" as is, while other fields can be obfuscated in varying degrees according to the desires of the data provider. The `sensitive-copy` table definition above could be re-written as follows:

```
<table name="sensitive-copy">
  <variable name="query" type="string">
    <iteration query="SELECT ID,Name,SSN,Degree,Salary FROM sensitive"/>
  </variable>
</table>
```
In the re-written SDDL above, the ID and Name fields have been eliminated in the copied table, and the SSN field has been obfuscated to a constant XXX-YY-ZZZZ. The Degree and Salary fields have been preserved at their original values. Of course, even revealing degree/salary information might yet be too sensitive for third-party release, and these could be further obfuscated still. See Chapter 7 on “Data Obfuscation” for a detailed discussion on data obfuscation degrees and methods.

2.5.5.2 Pool Iterations

A pool iteration causes the values of an associated column to iterate through the choices in a specified pool. A pool iteration is denoted by the use of the "pool" attribute in an iteration element. Pool iterations are useful for outputting pools as tables, among other things.

Suppose you were generating a university database, and one of your tables was to be college. Since there are a small number of colleges, the easiest thing might be to generate the table from a small pool. One could design the pool as follows:

```xml
<pool name="colleges">
  <choice name="Engineering"><ID>1</ID></choice>
  <choice name="Fine Arts"><ID>2</ID></choice>
  <choice name="Business"><ID>3</ID></choice>
  <choice name="Law"><ID>4</ID></choice>
</pool>
```
And one could generate a "college" table with the following SDDL code:

```xml
<table name="college">
  <field name="Name" type="string">
    <iteration pool="colleges"/>
  </field>
  <field name="ID" type="int">
    <formula>colleges[college].ID</formula>
  </field>
</table>
```

The resulting college table would look like:

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engineering</td>
<td>1</td>
</tr>
<tr>
<td>Fine Arts</td>
<td>2</td>
</tr>
<tr>
<td>Business</td>
<td>3</td>
</tr>
<tr>
<td>Law</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2-2: Results for Pool Iteration

### 2.5.5.3 Count Iterations

A count iteration causes a column value to iterate through a set of integers. A count iteration is denoted through the use of `base` and `count` attributes in an iteration element.

Consider the following field definition:

```xml
<field name="ID" type="int">
  <iterate base="101" count="10"/>
</field>
```

The ID column values would iterate between 101 and 110, inclusive.

Count iterations can be used to supply actual column values, but a more common use for them is to enforce the repetition of a task `count` times.
2.5.5.4 Cardinality in Iteration Elements

By default, iterations will produce exactly one output value for each member of the specified iteration set. However, it is possible to specify different behavior for an iteration. By using repeat specifiers, output cardinality can be changed. Consider the following SDDL snippet:

```xml
<field name="foo" type="int">
    <iteration base="1" count="5">
        <repeatMin>1</repeatMin>
        <repeatMax>5</repeatMax>
    </iteration>
</field>
```

In the example above, the `foo` field might be generated as follows: `{1,1,2,2,2,3,4,4,4,4,4,5,5}`. Each element of the iteration is repeated between 1 and 5 times, as requested. Iteration element cardinality can also be controlled through a statistical distribution, like this:

```xml
<field name="foo" type="int">
    <iteration base="1" count="5">
        <repeatDist>
            <tier prob="0.45" min="1" max="1"/>
            <tier prob="0.35" min="2" max="2"/>
            <tier prob="0.20" min="3" max="3"/>
        </repeatDist>
    </iteration>
</field>
```

The `foo` field, as described above, would iterate through the values `{1,2,3,4,5}`. Each value has a 45% chance of only being output once, a 35% chance of being output twice, and a 20% chance of being output 3 times.
2.5.5.5 Nesting Iterations

Iterations can be logically nested. The first iteration that appears in the table is considered the "outer" iteration, and ensuing iterations are "inner" iterations. Consider the following SDDL code:

```xml
<pool name="colors">
  <choice name="red"/>
  <choice name="green"/>
  <choice name="blue"/>
</pool>

<table name="nesting_example">
  <field name="A" type="int">
    <iteration base="1" count="3"/>
  </field>
  <field name="B" type="string">
    <iteration pool="colors"/>
  </field>
</table>
```

The resulting `nesting_example` table looks like this:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>red</td>
</tr>
<tr>
<td>1</td>
<td>green</td>
</tr>
<tr>
<td>1</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>red</td>
</tr>
<tr>
<td>2</td>
<td>green</td>
</tr>
<tr>
<td>2</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>red</td>
</tr>
<tr>
<td>3</td>
<td>green</td>
</tr>
<tr>
<td>3</td>
<td>blue</td>
</tr>
</tbody>
</table>

Table 2-3: Results for Nested Iterations
2.5.5.6 Iteration Variables

Certain data generation problems require the storing of state information between
the generation of rows. Each time the iteration element advances, the iteration variable is
initialized to the formula specified by the user.

Consider the following graph:

![Sample Directed Graph](image)

Figure 2-5: Sample Directed Graph

Can legal paths through this graph be generated? Yes. First, the graph needs to
be modeled as a pool:

```xml
<pool name="graph">
  <choice name="A"><next>D</next></choice>
  <choice name="B"><next>A</next></choice>
  <choice name="C"><next>B</next></choice>
  <choice name="D"><next>C</next></choice>
</pool>
```

Legal paths through the graph could then be produced by generating a 1-column
table as follows:

```xml
<table name="paths">
  <variable name="pathStart" type="string">
```

39
<iteration base="1" count="5">
  <repeatMin>2</repeatMin>
  <repeatMax>10</repeatMax>
  <!-- initialize to random graph node -->
  <itervar name="currNode"
    type="string"
    init="graph"/>
  <!-- initialize to empty string -->
  <itervar name="currPath"
    type="string"
    init=""/>
</iteration>
</variable>
</field>
</variable>
<field name="path" type="string">
  <formula>currPath = currPath + currNode</formula>
</field>
<variable name="updateNode" type="string">
  <formula>currNode = graph[currNode].next</formula>
</variable>
</table>

The pathStart variable provides for 5 "generation threads", each of which starts at an arbitrary point in the graph and generates successive paths from that point.

The only field (and therefore the only output column) contains the successively generated paths. Sample output might look like:

<table>
<thead>
<tr>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>AD</td>
</tr>
<tr>
<td>ADC</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>BA</td>
</tr>
<tr>
<td>BAD</td>
</tr>
<tr>
<td>BADC</td>
</tr>
<tr>
<td>BADCB</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>DC</td>
</tr>
<tr>
<td>DCBA</td>
</tr>
<tr>
<td>DCBAD</td>
</tr>
<tr>
<td>DCBADC</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>AD</td>
</tr>
</tbody>
</table>
This is a trivial example of how iteration variables might be useful in traversing graph-like models. It is easy to see how this example might be extended to include many more nodes, multiple fan-in and fan-out per node, and distances between nodes. The application chapters of this dissertation will showcase some more complicated, more useful examples of using iteration variables.

2.6 Summary

This chapter described the synthetic data description language (SDDL), which serves as the input language for SDG. SDDL provides constructs that cover all angles of data set description including:

- Intra-row dependencies are enforceable via formulas and pools.
- Loose (statistical or iterative) inter-row dependencies can be described with iterations, min/max constraints, statistical distributions, and weighted pools.
- Tight (matching) inter-row dependencies can be enforced with iteration variables.
- Inter-table dependencies can be enforced using query iterations or query pools.
Pools can be used as modeling tools. In this chapter, a pool was used to model a graph. In the application chapters of this dissertation, pools and iteration variables will be used to model much more complicated concepts.

SDDL compares well with other synthetic data input languages. The concept of (potentially nested) weighted pools allows SDDL to match the power of the input languages associated with [7] and [9]. While SDDL may lack some of DGL’s [13] ability to enforce complex statistical distributions, it does match DGL in terms of intra-row dependencies and the support of user-specified plug-in functions.

Two ideas combine to make SDDL unique:

- All SDDL constructs were designed to be run in parallel. This allows the SDG generation engine to take care of parallelization and partitioning “behind the scenes”; the user is not burdened with including parallelization details in the SDDL input file.

- The combination of pools and iteration variables allows SDDL to model some very complicated concepts, such as graphs and context-free languages. It is believed that this modeling power is unique to SDDL among synthetic data input languages.
3 SDG SOFTWARE ARCHITECTURE

The choice was made to implement the SDG data generation engine entirely in Java. This is a departure from most previous synthetic data generation engines, which were typically written in languages like C [5, 9], C++ [13] and C# [14]. Admittedly, a program written in Java will typically run slower than a functionally identical program written in C or C++ (although some would say that the gap is closing – see [20]). However, the effortless portability offered by Java made it attractive. And that portability has paid dividends; the generator has been run on many different platforms and operating systems without any problems. Also in its favor, Java provides some nice “pluggability” features that make it possible to link in new code “on-the-fly”, which makes it easier to support user-defined functions.

This chapter describes the design of SDG via UML diagrams and explanatory remarks. The top-level and generation classes are direct analogs to SDDL constructs. As such, they carry all of the information necessary to define and process their associated constructs. The parser classes handle the parsing and evaluation of formulas, pool references and queries. The random number generation classes contain the logic necessary to support multiple types of random number generators. The environment classes allow various objects (Pools, Fields, Constants, IterationVariables) to be grouped together and accessed in an orthogonal fashion. The SaxDBUnmarshaller class contains high-level logic for processing source SDDL files. The Globals class contains a small set of global variables used by SDG during execution. The Main class contains the main() method that is the starting
point for SDG execution. The chapter ends with some sequence diagrams that shed further light on the algorithms behind the architecture.

3.1 Top-Level Classes

The classes described in this section are high-level classes that contain basic organizational information corresponding to the contents of the source SDDL file. These classes are direct software analogs to SDDL elements, and will be described in terms of their SDDL counterparts.

3.1.1 Database Class

The outer-most element in an SDDL file is a database element. The Database class holds information about all of the elements in an SDDL database element.

```
Database
- m_tables: ArrayList<Table>
- m_pools: ArrayList<Pool>
- m_constantList: ArrayList<Constant>
- m_swathSize: long
- m_env: Environment
+
+addTable(t:Table): void
+findTable(name: String): Table
+addPool(p:Pool): void
+findPool(name: String): Pool
+addConstant(c:Constant): void
+findConstant(name: String): Constant
+getEnvironment(): Environment
+setSeed(s:long): void
+<<static>> openDBCConnection(): Connection
+generateData(): void
+writeSDGLFile(fname: String): void
```

Figure 3-1: Database Class
An SDDL database element contains one or more table elements, zero or more pool elements, zero or more constant elements, and an optional seed element. Likewise, the Database class contains lists of Table objects, Pool objects, and Constant objects (all to be described below). During the parsing of the source SDDL file, the addTable(), addPool(), and addConstant() methods are employed whenever child table, pool, or constant elements (respectively) are encountered.

The generateData() method will effect the generation of all data specified by the source SDDL file. It calls generation methods in each of its contained Table elements. During generation, the findTable(), findPool(), and findConstant() methods may be called to locate named entities.

The writeSDGLFile() method will export the contents of the Database object to an SDDL file with the specified file name. It is used by the graphical front-end to create SDDL files for users. This method will recursively call all of the toFile() methods associated with its various component objects.

### 3.1.2 Tables and Fields

The Table and Field classes (Figure 3-2) correspond to the table and field elements in SDDL. A Database object contains one or more Table objects, and a Table object contains one or more Field objects.

---

3 This method is a holdover from the days when SDDL was called SDGL.
During the parsing of the source SDDL file, Field objects are added to a parent Table object as appropriate using the addField() method of the parent Table object. During data generation, the findField() method may be called to locate a named field during formula evaluation.

A Table object contains all of the information necessary to generate a table from the information given in the source SDDL file. It contains a name and a set of fields. There are two generateData() methods: the first generates directly to a specified database, the second will generate to a file whose name is based on the table name.

The Table.toFile() method is used to export the contents of a Table object to a user-specified SDDL file.

A Field object stores information about how to generate a specific field for its parent Table. A Field object is instantiated with a name, a data type, and a pointer to the parent Table object. By default, the datumGenerator g is initialized to a simple generation object corresponding to the specified field type.

A Field object contains a datumGenerator object. The setMin(), setMax(), setStep() and generateValue() methods of a Field object simply call those same methods in the datumGenerator object. The setGenerator(), setDistribution(), setFormula(), setIteration(), and setQueryPool() methods simply replace “g”, the datumGenerator member object, with the datumGenerator-derived object passed as a parameter to those methods.
There are a number of `datumGenerator` classes, which will be described later. One important thing to note is that Strings are the common currency among all classes. While native values may be strings, ints, reals, dates, or booleans, all are converted to strings at the higher processing levels in order to adhere to a common
interface. Thus, the `generateValue()` method will return a `String` object, regardless of the type associated with the `Field` object.

The `Field.toFile()` method will serialize a `Field` object into the user-specified SDDL file.

### 3.1.3 Pools and PoolChoices

The `Pool` and `PoolChoice` classes (Figure 3-3) are the software analogs of the pool and choice elements in SDDL. A `Database` object contains zero or more pool objects, and a pool object contains one or more `PoolChoice` objects. A `PoolChoice` object can contain `Pool` objects and auxiliary data strings.

A `Pool` object stores information related to a specific pool element. During SDDL parsing, the `addChoice()` method of a `Pool` object is called to add child `PoolChoice` objects as they are encountered. The `getChoice()` method may be called during generation to return a named choice. The `generateValue()` method is called to return the name of a random `PoolChoice` object; if the `Pool` object is weighted (i.e., one of it’s `PoolChoice` elements has a weight other than the default of 1.0), the `generateValue()` method will call `generateWeightedValue()`.

The `Complete()` method is called by the SDDL parser to normalize the weights in a weighted pool. The `toFile()` method serializes a `Pool` object to SDDL format into a caller-specified file.
A PoolChoice object stores information related to a specific choice element. A PoolChoice object has a name (mandatory), zero or more sub-pools, and zero or more auxiliary data elements. A PoolChoice object has a default weight of 1.0; if the weight is set to anything other than 1.0, then the containing Pool object is considered weighted. The addSubPool() and addAuxData() methods are called during SDDL source file parsing as child pool or auxiliary data elements are encountered. The toFile() method serializes a PoolChoice object to a user-specified SDDL file.
3.1.4 Constants

In SDDL, a database element can contain zero or more constant elements. Constant elements are designed to hold constant values like PI, DAYS_PER_YEAR, or similar values. The Constant class contains information relative to an SDDL constant element.

The Constant class (Figure 3-4) derives from the StateVariable class, which derives from the Lexeme class (described later in Section 3.3). A Constant object is instantiated with name, data type and value parameters. Because Constant inherits from Lexeme, Constants can be used in formulas just like field names and numeric constants, with one difference: the Constant class overrides the assign() method from the StateVariable class such that any attempt to assign to a
Constant object will result in an Exception being thrown. The toFile() method serializes a Constant object into a caller-specified SDDL file.

3.2 Generation Classes

The classes in this section are designed to generate data. These classes all implement the valueGenerationInterface:

```java
public interface valueGenerationInterface
{
    public String generateValue(RngInterface r);
}
```

Each of these classes therefore has a generateValue() method which accepts an RngInterface object as a parameter and returns generated data in String format.

Recall that, in SDDL, there are five ways to constrain data for a field:

- min/max
- statistical distribution
- formula
- iteration
- queryPool

The min/max method is the most basic way to constrain a field, and min/max generation classes are considered simple generation classes. By default, Field objects instantiate a simple generator class corresponding to the specified field type during construction.

The generation classes dealing with distributions, formulas, iterations, and queryPools are considered complex generation classes.
3.2.1 Simple Generation Classes

The classes (Figure 3-5) in this subsection are simple min/max generators for specific data types.
The `datumGenerator` class is the abstract base class for all `datumGenerator`-derived classes. It has `minParseTree`, `maxParseTree`, and `stepParseTree` members for dealing with min, max, and step values specified as formulas. It has abstract `setMin()`, `setMax()` and `setStep()` methods for specifying min, max and step, which must be implemented by all derived classes. It has `getIntValue()`, `getDoubleValue()`, `getStringValue()`, `getDateValue()` and `getBoolValue()` methods which are used to access the latest-generated value in various formats; these must all be implemented by derived classes. The `fromType()` static method will instantiate the correct `datumGenerator`-derived class based on the given data type string. The `toFile()` method will serialize a `datumGenerator` (or `datumGenerator`-derived) object to a caller-specified SDDL file.

The remainder of the classes all correspond to the basic field types supported in SDDL. The following table shows the `datumGenerator`-derived class associated with each field type, as well as the default min/max values associated with them:

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Associated <code>datumGenerator</code>-derived Class</th>
<th>Default Minimum</th>
<th>Default Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td><code>intGenerator</code></td>
<td>1</td>
<td>1000000</td>
</tr>
<tr>
<td>real</td>
<td><code>realGenerator</code></td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>date</td>
<td><code>dateGenerator</code></td>
<td>2000-01-01</td>
<td>2010-01-01</td>
</tr>
<tr>
<td>time</td>
<td><code>timeGenerator</code></td>
<td>00:00:00</td>
<td>23:59:59</td>
</tr>
<tr>
<td>timestamp</td>
<td><code>timestampGenerator</code></td>
<td>2000-01-01</td>
<td>2009-12-31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00:00:00</td>
<td>23:59:59</td>
</tr>
<tr>
<td>bool</td>
<td><code>boolGenerator</code></td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 3-1: Type-to-Generator Associations

Consider the following SDDL field definition:
The $F1$ field object as defined above will contain an intGenerator upon instantiation, because it has a type of int. By default, the minimum is specified to be 1 and the maximum is specified to be 1000000, with a step-value of 1. The values generated for $F1$ would therefore be between 1 and 1000000, inclusive. However, if min and max values are specified in SDDL:

```xml
<field name="F1" type="int">
  <min>15</min>
  <max>30</max>
</field>
```

then the `setMin()` and `setMax()` methods of the intGenerator are called during SDDL parsing, and the values generated will be constrained to the specified range (in this case: 15 through 30, inclusive). The same general logic applies to the other datumGenerator-derived classes, except for stringGenerator.

The stringGenerator class is a special case. Specifying min/max for a stringGenerator results in a different kind of behavior than it would for the other simple generators. The length of the min string is used to determine the minimum length of the generated string; the left-most character in the min string denotes the minimum value for each individual character of the generated string. The length of the max string is used to determine the maximum length of the generated string; the left-most character in the max string denotes the maximum value for each individual character of the generated string.

For example, consider the following string field definition:

```xml
<field name="F2" type="CHAR(4)">
  <min>aa</min>
  <max>dddd</max>
</field>
```
The strings generated for field F2 above would be between two and four characters long, each character being between a and d, inclusive.

Min and max values can be specified via formula, instead of a simple constant. Formula objects will be described in the next subsection.

### 3.2.2 Complex Generation Classes

The generator classes in this section are the `datumGenerator`-derived classes that use some method other than `min/max` to constrain their generation values. Some of these build on top of the simple generator classes.

#### 3.2.2.1 Distributions and Tiers

The `Distribution` and `Tier` classes are used to handle information related to statistical distribution constraints. The following SDDL is an example of a field constrained with a statistical distribution constraint:

```xml
<field name="age" type="int">
  <dist>
    <tier prob="0.2" min="20" max="30"/>
    <tier prob="0.8" min="31" max="40"/>
  </dist>
</field>
```

In the example above, 20% of the ages generated would be between 20 and 30 (inclusive), and 80% of the ages generated would be between 31 and 40 (inclusive).

A `Distribution` object stores a number of `Tier` objects. Each `Tier` object has associated with it a probability, min, and max. When `generateValue()` is called
on the Distribution object, it will generate a random value between 0 and 1. Using
that value, it will select one of its Tier objects, and return the result from a call to
generateValue() on that Tier object. Using the example of the age field defined
above, suppose the generateValue() method in the Distribution object
produced a value of 0.25; the second tier would then be selected, and a value between
31 and 40 (inclusive) would be returned.

3.2.2.2 Formulas

A Formula object handles information related to formula constraints. The
following SDDL is an example of a formula constrained field:

```xml
<field name="ship_date" type="date">
  <formula>order_date+IRND(5)</formula>
</field>
```

In the example above, the ship_date field would be generated as anywhere
from 0 to 4 days after the order_date field.

Upon instantiation, a Formula object will parse its source formula and store the
resulting parse tree in parseTreeRoot. When generateValue() is called, the
Formula object will simply call evaluate() on parseTreeRoot. For more on
parsing and evaluation of formulas, see Section 3.3 on parser-related classes.

3.2.2.3 QueryPools

A QueryPool object will maintain information relative to a queryPool
constraint. The following SDDL shows an example of field with a queryPool constraint:

```xml
<field name="name" type="CHAR(24)">
  <queryPool query="SELECT name from students
```
WHERE age = [ageField]"/>

Upon instantiation, the QueryPool object will parse its source query and store the resulting parse tree in parseRoot. A non-parameterized query like SELECT name FROM students will be stored as a one-lexeme parse tree containing a string constant. A parameterized query such as the one in the example above will be parsed into its component parts and stored as a parse tree. When generateValue() is called, the parse tree in parseRoot will be evaluated, and the resulting string will be sent in the form of a query to the reference database. For more information about query parsing and evaluation, see the next section on parser-related classes.
3.2.2.4 Iterations

The Iteration classes track information relative to iteration constraints. The Iteration base class holds all data members common to all flavors of iteration: a repeat count generator (rptGen), a list of iteration variables (variableList), the list of items through which to iterate (items), and some accounting variables (currIdx, prevIdx, rptCount, prev, next). The repeat count generator defaults to outputting each element once, but can be manipulated through SDDL repeatDist and
repeatMin/repeatMax elements. The currIdx and prevIdx variables keep track of the current and previous output position in the items array. The prev and next members allow for the maintenance of Iteration object chains. The rptCount variable downcounts the remaining times that the item at currIdx should be output. The base class also provides the iterVarsToFile() and repeatSpecToFile() methods to serialize the iteration variables and repeat specification to an SDDL file.

The generateRepeatCount() method will generate a repeat count for each element of the iteration (i.e., how many times the element should be output through generateValue). Repeat count defaults to 1; other values can be specified via either <min> and <max> elements or a <repeatDist> element within an iteration. The base Iteration class also provides default methods bump() for progressing the iteration and generateValue() for returning the next iteration element. Finally, the base Iteration class specifies an abstract regenerate() method that must be implemented by all Iteration-derived classes. The regenerate() method is used to populate the items array, and is implemented differently by each Iteration-derived class.
Simplistically put, the `generateValue()` method is called once for each row output by the containing `Table` object. After outputting each row, the `Table` object will call `bump()` on the inner-most iteration.

Figure 3-8 details the logic behind a call to the `Iteration.generateValue()` method:

- The `currIdx` variable will be initialized to -1, and will also be set to -1 when the iteration “wraps”. In either case, it is necessary to regenerate a repeat count, regenerate the iteration values, re-initialize the iteration variables, and set the `currIdx` to the beginning index for this generation.
process (index_base depends upon the index of this generation process).

- If currIdx has been incremented since the last call to
  generateValue(), it will be necessary to regenerate the repeat count,
  re-initialize the iteration variables, and set prevIdx = currIdx.

- The value returned is items[currIdx].

- If the repeat count was generated as 0, then suppress the output of this row.

Figure 3-9 details the logic behind a call to the Iteration.bump() method:

- The repeat count is decremented.

- If the repeat count is less than 1, then:
  
  o The currIdx must be incremented (index_skip depends upon the
    number of processes participating in the generation).

  o If currIdx is incremented past the size of the items array, then
    “bump” the next-innermost Iteration object (if one exists) and set
    currIdx = -1.
Figure 3-8: Iteration.generateValue() Logic
3.2.2.4.1 Count Iterations

A CountIteration contains information relative to an SDDL count iteration constraint. The following SDDL code is an example of a count iteration-constrained field:

```xml
<field name="ID" type="int">
  <iteration base="1" count="10"/>
</field>
```

One can specify base and count parameters via constants (as in the example above) or formulas. When formulas are used, they are parsed and their trees are stored in baseParseRoot and countParseRoot; when constants are used, they are stored in the base and count members. The CountIteration implements the
regenerate() method to build an array of strings corresponding to the specified base and count; in the example above, the regenerate() method would produce \{"1","2","3",...,"10"\}.

### 3.2.2.4.2 Pool Iterations

A PoolIteration contains information relative to an SDDL pool iteration constraint. The following SDDL code is an example of a pool iteration-constrained field:

```sddl
<pool name="color">
  <choice name="red"/>
  <choice name="green"/>
  <choice name="blue"/>
</pool>
...
<field name="colors" type="string">
  <iteration pool="color"/>
</field>
```

Upon instantiation, a PoolIteration object will store its source pool reference in parseRoot. PoolIteration provides a regenerate() method that will populate the items array with the choices in the user-specified pool. In the example above, regenerate() would populate the items array with \{"red","green","blue"\}.

### 3.2.2.4.3 Query Iterations

QueryIterations track information relative to a query iteration. The following SDDL code is an example of a query iteration-constrained field:

```sddl
<field name="StudentID" type="int">
  <iteration query="SELECT StudentID, StudentName FROM Students"/>
</field>
```
QueryIterations are more complex than the other iteration classes. The regenerate() method will actually populate an SQL ResultSet, instead of an ArrayList of Strings. The generateValue() method will return values from the left-most column in the ResultSet, but the remaining columns are accessible through applying the subField ($) operation to the query iteration field:

```xml
<field name="StudentName" type="CHAR(32)">
  <formula>StudentID$StudentName</formula>
</field>
```

Because of these differences, the QueryIteration class overrides the generateValue() and bump() methods of the Iteration base class. The generateValue() method takes its result from the ResultSet, and the bump() method maintains the ResultSet instead of the items list.

### 3.3 Parser-related Classes

The classes (Figure 3-10) in this section support the parsing and evaluation of formulas, pool references, and queries. SDG employs a recursive descent parser to handle these duties. The actual design and techniques used to implement the parser borrow heavily from pseudocode provided in an “Advanced Programming Languages” course taught by Dr. John Lusth at the University of Arkansas in Spring 2005.
SDG uses a parse-once, evaluate-often scheme for processing formulas and queries. Formulas and queries will be parsed into parse trees as they are read from the source SDDL file(s). These parse trees will be evaluated as data is actually generated.

The Lexer class methods will break a formula or query into its component Lexemes and return a LexemeQueue containing those Lexemes. The Parser class provides recursive-descent parser methods to convert the Lexemes in a LexemeQueue into a parse tree. A parse tree is made up of Lexemes, including the root of the parse tree. Calling evaluate() on the root Lexeme of a parse tree will result in the parse
tree being evaluated and will return a Lexeme containing an evaluated value for the parse tree.

The Lexeme class provides an evaluate() method which will return the evaluation result for the Lexeme, whether it is a singleton or the root of a tree. This class contains the evaluation logic used in the processing of formulas and queries.

The internal Lexeme types supported by SDG are enumerated in the following table (entries in **BOLD** represent variables or complex operations):

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>PERIOD (‘.’)</th>
<th>CPAAREN (‘)’)</th>
<th>LT (‘&lt;’)</th>
<th>COMMA (‘,’)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>OBRACKET (‘{’)</td>
<td>INDEX</td>
<td>GE (‘&gt;=’)</td>
<td>CALL</td>
</tr>
<tr>
<td>STRING</td>
<td>CBRACKET (‘}’)</td>
<td>DEREF</td>
<td>LE (‘&lt;=’)</td>
<td>POOLREF</td>
</tr>
<tr>
<td>DATE</td>
<td>PLUS (‘+’)</td>
<td>CHOICE</td>
<td>TERNARY (‘?’)</td>
<td>STATEVAR</td>
</tr>
<tr>
<td>TIME</td>
<td>MINUS (‘-’)</td>
<td>POOL</td>
<td>CONDITION (‘:’)</td>
<td>ASSIGN (‘=’)</td>
</tr>
<tr>
<td>NAME</td>
<td>MULTIPLY (‘*’)</td>
<td>EQ (‘==’)</td>
<td>LOGICAL (‘</td>
<td>’)</td>
</tr>
<tr>
<td>BOOL</td>
<td>DIVIDE (‘/’)</td>
<td>NEQ (‘!=’)</td>
<td>LAND (‘&amp;&amp;’)</td>
<td>MODULUS (‘%’)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>OPAREN (‘(’)</td>
<td>GT (‘&gt;’)</td>
<td>LNOT (‘!’)</td>
<td></td>
</tr>
</tbody>
</table>

See Appendix B for a complete description of formula expression grammar and semantic rules.

### 3.3.1 Formula Processing

Consider the following SDDL field definition:

```xml
<field name="F2" type="int">
  <formula>F1*10+4</formula>
</field>
```

During the parsing phase, the **Parser** object will first call the **Lexer.lexFormula()** method on the input formula. The **Lexer** will return a **LexemeQueue** composed of the following Lexemes: **{NAME:F1, MULTIPLY,**

67
The Parser will then translate the

INTEGER:10, PLUS, INTEGER:4}.

LexemeQueue into a parse tree using the Parser.parseFormula() method:

Figure 3-11: Parse Tree for Formula F1*10+4

Each time that generateValue() is called for Field F2, the parse tree above
will be evaluated into a generation value.

3.3.2

Query Processing
Consider the following SDDL field definition:

<field name=”F3” type=”int”>
<queryPool query=”SELECT name,salary
FROM employees
WHERE age = [F2]
ORDER BY salary”/>
</field>
During

the

parsing

phase,

the

Parser

Lexer.lexQuery() method on the input query.
LexemeQueue

composed

of

the

following

object

will

first

call

the

The Lexer will return a

Lexemes:

{STRING:“SELECT

name,salary FROM employees WHERE age = ”, PLUS, NAME:F2,

68


PLUS, STRING:” order by salary”). The Parser will then translate the LexemeQueue into a parse tree using the Parser.parseQuery() method:

![Parse Tree for Query](image)

Figure 3-12: Parse Tree for Query "SELECT name, salary FROM employees
WHERE age = [F2] ORDER BY salary"

Each time that generateValue() is called for Field F3, the parse tree above will be evaluated into a generation value.

### 3.3.3 Pool Reference Processing

Although pool references are technically parts of formulas, they are considered separately here as they do entail some special processing. Consider the following SDDL field definition:

```xml
<field name="F4" type="int">
  <formula>
    0+states[state].cities["New York"].population
  </formula>
</field>
```

The Lexer.lexFormula() method would initially identify the following Lexemes:{INTEGER:0,PLUS, STRING:”states[state].cities[“New York“].population”}. Complex pool references contain periods, and the formula lexer cannot specifically look for periods, because periods are also contained in real numbers (as decimal points), and
lexing down to the period level would greatly complicate the logic for processing real numbers. Instead, the formula lexer will examine each primary looking for the presence of an open-bracket ([]) character. If the formula parser finds an open-bracket character in a primary, then that primary is further parsed as a pool reference, and the resulting tokens are inserted into the LexemeQueue. In this case, the final LexemeQueue constructed in Lexer.lexFormula() would look like: `{INTEGER:0, PLUS, NAME:states, OBRACKET, NAME:state, CBRACKET, PERIOD, NAME:cities, OBRACKET, STRING:"New York", CBRACKET, PERIOD, NAME:population}. The Parser.parseFormula() method would convert that LexemeQueue to a parse tree:

![Parse Tree](image)

Figure 3-13: Parse Tree for Formula $0 + \text{states}[\text{state}].\text{cities}[\text{"New York"]}.\text{population}$

Each time that `generateValue()` is called for Field F4, the parse tree above will be evaluated into a generation value.
An even more complex process is used to lex and parse the following:

\[
\text{<field name="F5" type="string">}
\text{states["New" + "York"] + 5.cities</field>}
\]

The formula above can be interpreted as “pick a city from the entry 5 away from ‘New York’ in the states pool”. Pool indexers can be arbitrary expressions (in this case, “New” + “York”), and the addition (“+”) operator applied to a PoolChoice results in “shifting” that choice. The formula above would initially be lexed as:

\[
\{\text{Name:states[, STRING:"New", PLUS, STRING:" ", PLUS, STRING:"York", NAME:], PLUS, INTEGER:5, NAME:.cities}\}.
\]

A second round of lexing would look for open-brackets, close-brackets, and periods, resulting in:

\[
\{\text{Name:states, OBRACKET, STRING:"New", PLUS, STRING:" ", PLUS, STRING:"York", OBRACKET, PLUS, INTEGER:5, PERIOD, NAME:cities}\}.
\]

That LexemeQueue would finally be parsed as:

![Parse tree for formula states[“New” + “York”] + 5.cities](image)

Figure 3-14: Parse tree for formula states[“New” + “York”] + 5.cities
3.4 Random Number Generation Classes

Figure 3-15 shows the classes involved with random number generation. At present, SDG supports two options for random number generation: The LinearRng class, which is built on top of the Java random number generator (which is a linear congruent generator, or LCG), and the CMRGRng class, which is built around the RngStream class (which is a combined multiple recursive generator, or CMRG). Both classes implement the RngInterface class, which allows for them to be used interchangeably in the code.

The RngFactory class is used to create a random number generation class. The createRng() method will return a LinearRng object (which is the default random number generator), and the createRng(String rngName) method will return either a LinearRng object or a CMRGRng object, depending upon the value of the “rngName” parameter.

The “-rng:<RNGNAME>” command line switch governs which of these two generators will be used by SDG to perform data generation. By default, LinearRng will be used.
3.5 Environment Classes

Figure 3-16 shows the classes involved in implementing environments. An Environment object tracks named environment variables valid in a certain scope. A Database object will track Database-scoped objects (Pools, Constants), and a Table object will track Table-scoped objects (Fields, IterationVariables). An Environment object can have a parent Environment object, allowing for the
tracking of nested environments. For example, a Table Environment object will always contain a “parent” pointer to the Database Environment object, so that Database environment elements will be visible from the Table object.

Each item in an Environment object implements the EnvironmentElementInterface. Currently, StateVariables, Constants, Pools and Fields all implement EnvironmentElementInterface; this abstraction streamlines the process of Lexeme evaluation.

```
Environment

-m_elementHash: HashMap<String,EnvironmentElementInterface>
-m_parent: Environment
+Environment()
+Environment(parent:Environment)
+addElement(e:EnvironmentElementInterface): EnvironmentElementInterface
+find(name: String): EnvironmentElementInterface

<<interface>>
EnvironmentElementInterface

+TYPE_INTEGER: static int = 1
+TYPE_REAL: static int = 2
+TYPE_STRING: static int = 3
+TYPE_DATE: static int = 4
+TYPE_TIME: static int = 5
+TYPE_BOOL: static int = 6
+TYPE_TIMESTAMP: static int = 7

+getName(): String
+getTypeNumber(): int
+getStringValue(): String
+getIntValue(): int
+getRealValue(): double
+getBoolValue(): boolean
+getDateValue(): java.util.Date
+assign(assignRoot: Lexeme): Lexeme
+evaluate(): Lexeme
```

Figure 3-16: Environment Classes
3.6 Plugin Function Interface

SDG and SDDL support user-specified plugin functions. All plugin functions implement the `PluginFunctionInterface` shown in Figure 3-17. Any number of parameters (including 0) can be passed via the `params` parameter, the evaluation environment is passed in the `env` parameter, and the system-wide random number generator is passed in the `rnd` parameter.

![Figure 3-17: PluginFunctionInterface](image)

To implement a plugin function called `Foo()`, one would first create a file called “Foo.java”, and then populate it as follows:

```java
public class Foo implements sdg.PluginFunctionInterface {
    public Lexeme call(ArrayList<Lexeme> params, Environment env, RngInterface rnd) {
        ...
    }
}
```

For a more detailed example of a plugin function, see section 2.2.
3.7 The SaxDBUnmarshaller Class

The SaxDBUnmarshaller class contains the logic to convert a source SDDL file into a Database object. It inherits from the org.xml.sax.helpers.DefaultHandler class, and implements the characters(), startElement() and endElement() methods associated with that class. The startElement() method contains logic to handle the start of an SDDL element, the endElement() method contains logic to handle the end of an SDDL element, and the characters() method contains logic to handle character data within an SDDL element.

The parse() methods will parse an input SDDL file (given as either a file name or a File object) into a Database object. The resulting Database object is accessible via the getDB() method.

General rules for SDDL processing are as follows:

- The startElement() method, which reacts to start tags, will do one of two things. For empty elements like <formula>, <min> and <max>, nothing will be put onto the stack. Instead, the string assembled by the characters() method will be put onto the stack later. For non-empty elements, a new object of a type associated with the start tag will be instantiated and added to the stack.

- The endElement() method, which reacts to end tags, will pop its associated object off of the stack and associate that element with the element now on top of the stack.
More specific SDDL parsing element rules will be given in some of the sequence diagrams at the end of this chapter.

Figure 3-18: SaxDBUnmarshaler Class

In both parse() methods, the following code is executed (line numbers added for explanatory purposes:

1. SAXParserFactory factory = SAXParserFactory.newInstance();
2. SAXParser saxParser = factory.newSAXParser();
3. XMLReader parser = saxParser.getXMLReader();
4. parser.setContentHandler(this);
5. parser.parse(src); // src is InputSource associated with input SDDL file
The `setContentHandler()` call in line 4 sets the content handler for the `XMLReader` to the `SaxDBUnmarshaller` object. The `parser.parse()` call in line 5 will parse the input SDDL file. As it does, it will call the `startElement()`, `endElement()`, and `characters()` methods of the `SaxDBUnmarshaller` class as it encounters these situations.

### 3.8 The Globals Class

The `Globals` class holds a number of global values used during the execution of the synthetic data generator.

<table>
<thead>
<tr>
<th>Globals</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+nprocs: static int = 1</code></td>
</tr>
<tr>
<td><code>+myproc: static int = 0</code></td>
</tr>
<tr>
<td><code>+fileMode: static boolean = false</code></td>
</tr>
<tr>
<td><code>+swathSize: static int = 100</code></td>
</tr>
<tr>
<td><code>+setUpMode: static boolean = false</code></td>
</tr>
<tr>
<td><code>+inFileName: static String = &quot;&quot;</code></td>
</tr>
<tr>
<td><code>+seed: static long = 0</code></td>
</tr>
<tr>
<td><code>+dbIsOracle: static boolean = false</code></td>
</tr>
<tr>
<td><code>+dbIsODBC: static boolean = true</code></td>
</tr>
<tr>
<td><code>+dbIsMysql: static boolean = false</code></td>
</tr>
<tr>
<td><code>+dbName: static String = null</code></td>
</tr>
<tr>
<td><code>+db: static Database = null</code></td>
</tr>
<tr>
<td><code>+end: static RngInterface = new RngFactory.create()</code></td>
</tr>
<tr>
<td><code>+fieldSep: static char = ','</code></td>
</tr>
<tr>
<td><code>+suppressRow: static boolean = false</code></td>
</tr>
<tr>
<td><code>+jitGeneration: static boolean = false</code></td>
</tr>
<tr>
<td><code>+useQuotes: static boolean = true</code></td>
</tr>
<tr>
<td><code>+quoteChar: static char = '\'</code></td>
</tr>
</tbody>
</table>

`+<static>> SuppressThisRow(): void`
The `SuppressThisRow()` method will cause the row currently being generated to NOT be output. This method only causes the current row to be suppressed, and must be called again to suppress any additional rows.

The member variables of the `Globals` class are summarized in this table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nprocs</td>
<td>1</td>
<td>Number of processes participating in data generation. It is set to the value given by the –nprocs command line argument.</td>
</tr>
<tr>
<td>myproc</td>
<td>0</td>
<td>My generation process index, 0-based. This is set to the value given by the –procmnum command line argument.</td>
</tr>
<tr>
<td>fileMode</td>
<td>false</td>
<td>If true, generate data to text file instead of directly to database. Set to true by the –textout command line switch.</td>
</tr>
<tr>
<td>swathsize</td>
<td>100</td>
<td>Swath size to use in type 1 data generation (i.e., no iterations present in table). Each block of generated data will have this many rows. At present, this cannot be set to any other value by the user.</td>
</tr>
<tr>
<td>setupMode</td>
<td>false</td>
<td>If true, generate SQL setup files, but not actual data. Set to true by –setup command line switch.</td>
</tr>
<tr>
<td>inFileName</td>
<td>“”</td>
<td>Name of source SDDL file. Provided on command line.</td>
</tr>
<tr>
<td>seed</td>
<td>0</td>
<td>User-specified seed for random number generation. Specified via the &lt;seed&gt; SDDL element.</td>
</tr>
<tr>
<td>dbIsOracle</td>
<td>false</td>
<td>If true, SDG is generating directly to an Oracle database. Set to true via the -dbtype:oracle command line switch.</td>
</tr>
<tr>
<td>dbIsODBC</td>
<td>true</td>
<td>If true, SDG is generating directly to an ODBC database. Set to false by either the –dbtype:oracle, –dbtype:mysql or –textout command line switches.</td>
</tr>
<tr>
<td>dbIsMysql</td>
<td>false</td>
<td>If true, then SDG is generating directly to a MySQL database. Set to true by the –dbtype:mysql command line switch.</td>
</tr>
<tr>
<td>dbName</td>
<td>null</td>
<td>If generating directly to database, the name of the database to which we are generating. Set via the –dbname command line argument.</td>
</tr>
<tr>
<td>db</td>
<td>null</td>
<td>The Database object built through the parsing of the source SDDL file. Set during SDDL parsing.</td>
</tr>
<tr>
<td>rnd</td>
<td>New LinearRng</td>
<td>The global RngInterface object to be used for</td>
</tr>
</tbody>
</table>
randomization. This defaults to a LinearRng object, but can be assigned a CMRGRng object via the “-rng:CMRG” command line switch.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fieldSep</td>
<td>‘,’</td>
<td>Character used for field separation in output text files. Can be set via the SDDL &lt;fieldSep&gt; element.</td>
</tr>
<tr>
<td>suppressRow</td>
<td>false</td>
<td>Suppress the current row. Set to false at the start of each row generation, set to true by SuppressThisRow() method.</td>
</tr>
<tr>
<td>jitGeneration</td>
<td>false</td>
<td>If true, tables will be generated to text files as their definitions are parsed. If false, table generation will be delayed until all table definitions are parsed in.</td>
</tr>
<tr>
<td>useQuotes</td>
<td>true</td>
<td>If true, string and date data will be quoted when it is output. This field will be true unless the –q0 switch is specified on the command line.</td>
</tr>
<tr>
<td>quoteChar</td>
<td>‘’</td>
<td>If useQuotes is true, the quoteChar value stores the type of quote (single- or double-quote) with which to surround string and date data in the output. The presence of a –q1 switch on the command line will set quoteChar to a single-quote, and the presence of a –q2 switch on the command line will set quoteChar to a double-quote.</td>
</tr>
</tbody>
</table>

Table 3-2: Description of Globals Class Member Variables

### 3.9 The Main Class

The Main class provides the main() method, which is the entry point into the SDG application.

```java
Main
+Main() 
+<static> interpretArgs(args: String[]): void
+<static> main(args: String[]): void
```

Figure 3-20: Main Class
The interpretArgs() method will parse the command line switches and arguments passed by the user. The command line switches and arguments supported by SDG are:

- **“-nprocs:<XX>”** : specifies the number of processes participating in data generation. Defaults to 1.
- **“-procnum:<XX>”** : specifies the index of this generation process (0-based). Defaults to 0.
- **“-textout”** : specifies that the data should be output to a text file. By default, data is output directly to a database.
- **“-dbtype:<XX>”** : in the case of direct-to-database generation, specifies the type of database to which the application is generating data. Legal values for XX are “oracle”, “odbc” or “mysql” (not case sensitive). “odbc” is the default.
- **“-dbname:<XX>”** : in the case of direct-to-database generation, specifies the name of the database/schema to which the application should generate data.
- **“-setup”**: specifies that SDG should generate support files (.ctl oracle control file, .sql oracle setup file, .mctl mysql setup and control file) instead of actual data.
- **“-jitg”**: specifies “just-in-time generation”, where a tables are generated as soon as they are parsed. In the absence of this switch, table generation will be delayed until all tables are read in.
• “-rng:<RNGNAME>”: specifies which random number generator to use. If <RNGNAME> is “LCG”, then a LinearRng object will be instantiated for randomization. If <RNGNAME> is “CMRG”, then a CMRGRng object will be instantiated for randomization. If nothing is specified, SDG defaults to using LinearRng.

• -q[012]: specifies information about the manner in which string, date and time values are output. The –q0 flag specifies “No quoting of string/date/time values”. The –q1 flag specifies “Surround string/date/time values with single quotes”. The –q2 flag specifies “Surround string/date/time values with double quotes”.

3.10 Selected Sequence Diagrams

3.10.1 Program Execution

The main() method, after interpreting the command line arguments, will instantiate a SaxDBUnmarshalller object, and will use it to parse the source SDDL file. After the SDDL parsing has completed successfully, the main() method will extract the newly created Database object from the SaxDBUnmarshaller object, via the getDB() method. The main() method will then call generateData() on the Database object.

(Note: If the “-jitg” (for “just-in-time generation”) switch is specified on the command line, then each table will be generated as soon as it is parsed, rather than the default mode of parsing all tables before beginning generation.)
3.10.2 Parsing a Pool Element

Consider the following SDDL pool definition:

```sddl
<pool name="States">
  <choice name="AR">
    <pool name="Cities">
      <choice name="Bentonville"/>
      <choice name="Fayetteville"/>
    </pool>
  </choice>
  <Nickname>Natural State</Nickname>
</pool>
```

The pool definition above is contrived; in reality, you rarely see a one-choice pool ("States" in this case). But it serves to help illustrate the actions taken in parsing a pool.
definition, as it contains a nested pool ("Cities") and an auxiliary data item ("Nickname").

The following diagram shows the sequence of operations that occurs when the pool defined above is parsed:

![Sequence Diagram for Parsing a Pool](image)

**Figure 3-22: Sequence Diagram for Parsing a Pool**

The following table gives some general rules for parsing pool elements:
### 3.10.3 Parsing a Table Element

Consider the following SDDL table definition:

```xml
<table name="A" length="10">
  <field name="X" type="int">
    <min>1</min>
    <max>100</max>
  </field>
</table>
```

The following sequence diagram shows how the table definition above is parsed into a `Table` object:
Figure 3-23: Sequence Diagram for Parsing a Table

The following table gives some general rules for parsing table elements:

<table>
<thead>
<tr>
<th>Tag Type</th>
<th>startElement()</th>
<th>characters()</th>
<th>endElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Instantiate Table object and push onto stack.</td>
<td>NA</td>
<td>Pop Table object from stack and add to parent Database object.</td>
</tr>
<tr>
<td>Field</td>
<td>Instantiate a Field</td>
<td>NA</td>
<td>Pop Field object from</td>
</tr>
<tr>
<td>Key</td>
<td>Description</td>
<td>Action 1</td>
<td>Action 2</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------</td>
<td>---------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>min</td>
<td>No action (empty element).</td>
<td>Push value string onto stack.</td>
<td>Pop value string from stack, call <code>setMin()</code> on parent <code>Field</code> object, passing value string as parameter.</td>
</tr>
<tr>
<td>max</td>
<td>No action (empty element).</td>
<td>Push value string onto stack.</td>
<td>Pop value string from stack, call <code>setMax()</code> on parent <code>Field</code> object, passing value string as parameter.</td>
</tr>
<tr>
<td>Dist</td>
<td>Instantiate Distribution object and push onto stack.</td>
<td>NA</td>
<td>Pop Distribution object from stack and use as parameter to <code>setDistribution()</code> method in parent <code>Field</code> object.</td>
</tr>
<tr>
<td>Tier</td>
<td>Instantiate Tier object and push onto stack.</td>
<td>NA</td>
<td>Pop Tier object from stack and add to parent <code>Distribution</code> object.</td>
</tr>
<tr>
<td>Iteration</td>
<td>Instantiate Iteration object and push onto stack.</td>
<td>NA</td>
<td>Pop Iteration object from stack, call <code>setIteration()</code> on parent <code>Field</code> object.</td>
</tr>
<tr>
<td>Itervar</td>
<td>Instantiate IterationVariable object and push onto stack.</td>
<td>NA</td>
<td>Pop IterationVariable object from stack and add to parent <code>Iteration</code> object.</td>
</tr>
<tr>
<td>queryPool</td>
<td>Instantiate QueryPool object and push onto stack.</td>
<td>NA</td>
<td>Pop QueryPool object from stack, call <code>setQueryPool()</code> on parent <code>Field</code> object.</td>
</tr>
<tr>
<td>Formula</td>
<td>No action (empty element)</td>
<td>Push value string onto stack.</td>
<td>Pop value string from stack, instantiate <code>Formula</code> object from value string, call <code>setFormula()</code> on parent <code>Field</code> object.</td>
</tr>
</tbody>
</table>
3.10.4 Parsing a Formula Field

Formula parsing is complicated, and so a formula field example is included here. Consider the following SDDL field definition:

```
<field name="C" type="int">
  <formula>A+16*B</formula>
</field>
```

The sequence diagram that follows details the actions that are taken during the parsing of the field definition above. Upon detecting the end-tag for a formula element (</formula>), the parser will create a Formula object, passing the desired formula (in this case, “A+16*B”) to the Formula constructor. The Formula constructor will construct a Parser object; the Parser constructor will construct a Lexer object. The Formula constructor then calls parseFormula() on the Parser object. The Parser object will first lex the formula into an array of lexemes, then call expression() to convert the string of lexemes into an expression tree. The Formula object will store the resulting expression tree for later processing.

The same general sequence applies when formulas are used in the description of other constraint types, like min/maxes and statistical distributions.
3.10.5 Generating a Table

Consider the following SDDL table definition:

```
<table name="X" length="10">
  <field name="A" type="date">
    <min>1990-01-01</min>
    <max>1999-12-31</max>
  </field>
  <field name="B" type="date">
    <formula>A+IRND(10)</formula>
  </field>
</table>
```
What actions are taken to generate the data from the definition above? The answer is in the following sequence diagram:

Figure 3-25: Sequence Diagram for Generating Data

The sequence diagram above assumes that the user has provided the \texttt{-textout} command line switch while executing the data generator, so that data will be directed to a text file.

The Database.generateData() file will call either generateDataFile() or generateDataDB(), depending upon the presence of the \texttt{-textout} flag; in this case, generateDataFile() will be called. Database.generateFile() will call Table.generateData() for each
Table object contained in the Database object. The Table.generateData() method calls generateSQLInsertValueSet() for each row of data to be generated (in this case, 10 times for 10 rows). The generateSQLInsertValueSet() method calls generateValue() for each Field object contained in the Table object, then bundles the results into a String separated by a user-specified field separation character (default = ‘,’). The generateValue() method for a Field object will return the result from the generateValue() call on the Field’s datumGenerator; for field A, this is a dateGenerator, and for field B, it is a Formula. Each row is output to file.

The sequence diagram above hides some of the details of data generation. For one thing, generating data directly to a database is slightly more complicated. Also, the Table.generateData() method has to choose between 3 different algorithms for generating data:

- generationAlgorithm1(): used when no iteration-constrained fields are present in the table
- generationAlgorithm2(): used when iteration-constrained fields are present in the table, but there are no autoIncremented fields.
- generationAlgorithm3(): used when there are both iteration-constrained fields and autoIncremented fields present in the table.

These generation algorithms will be explored in more detail in the next chapter.
4 PARALLEL DATA GENERATION

In order to achieve maximal data generation rates, it was important to be able to leverage modern cluster computing technology and generate data in parallel. However, parallelism presents a number of challenges. Could data sets be generated identically (with respect to a constant SDDL input) from run to run regardless of the degree of parallelism? This property of determinism is a requirement for regression testing applications. Also, how much inter-process communication would be necessary during parallel data generation? Too much communication overhead would diminish the speed advantage gained through parallel execution.

The literature contains references to previous efforts to generate synthetic data in parallel. Gray’s work [5] provided techniques for parallelizing special-purpose data generators. However, Gray’s work lacked a general-purpose language for describing generated data. The MUDD generator [9] was capable of deterministic parallelism and included a general-purpose input language. However, that input language was only useful for specifying step-function distributions, and did not support the complex inter- and intra-row dependency mechanisms associated with SDDL. The KRDataGeneration generator [14, 15] claims the capability for distributed data generation, but provided little documentation to support their claim.

In the case of SDG, algorithms which can run in parallel have been developed to support almost all of the rich functionality provided in the SDDL input language. At the same time, this parallel logic generates consistent output (given consistent input) regardless of the degree of parallelism employed during data generation. Parallel support and partitioning are built into the generation engine itself; users specify that they want
parallel distribution but do not need to include generation distribution details within an SDDL file when describing the data they want generated.

This chapter will first discuss the algorithms that are employed for parallel (and sequential) data generation. At the end of the chapter, iteration variables are discussed. Iteration variables allow for tight inter-row data dependencies without reducing determinism or requiring inter-process synchronization.

4.1 Parallel Generation Algorithms

In the SDG architecture, one of the goals was to eliminate inter-process communication between parallel generation processes. In support of this goal, each generation process is assigned an index and a process count when it is launched; “you are process 0 of N, you are process 1 of N, …, you are process (N-1) of N.” Internally, each generation process will generate a portion of the data, which portion depends upon the process’ index and the total process count.

Figure 4-1: Parallel Data Generation
How does each process determine the portion of data that it will generate? In other words, how is the data partitioned among the generation processes? The answer depends upon the table description(s) contained in the input SDDL file. When generating a table without any iteration-constrained fields in it, generation algorithm 1 is used. When generating a table with iteration-constrained fields in it, generation algorithm 2 is used. And, in the unlikely case where iterations and auto-increments are present in the same table, generation algorithm 3 is used.

4.1.1 Generation Algorithm 1

Generation algorithm 1 is employed by SDG to generate data for tables without iterations. With no iterations present, a table’s row structure is predictable, and its rows are divided into “swaths”, each of size SWATHSIZE. The generation processes generate swaths in an alternating round-robin fashion. Before generating a swath, the generation process will call the re-seed function RF(seed, row) (where seed is the user-specified seed, and row is the start row of the swath), which uses seed and row to re-seed the random number generator.

In general, process P (0-based) of N will take the following actions (assuming a SWATHSIZE of S):

\[
\begin{align*}
RF(\text{seed}, \ P*S) \\
\text{Generate rows } P*S \text{ through } (P+1)*S-1 \\
RF(\text{seed}, \ (P+N)*S) \\
\text{Generate rows } (P+N)*S \text{ through } (P+N+1)*S-1 \\
RF(\text{seed}, \ (P+2N)*S) \\
\text{Generate rows } (P+2N)*S \text{ through } (P+2N+1)*S-1 \\
\end{align*}
\]
If there is only one generation process, and $SWATHSIZE = 100$, generation proceeds as follows:

Process 0 of 1:
RF(seed,0)
Generate rows 0-99
RF(seed,100)
Generate rows 100-199
RF(seed,200)
Generate rows 200-299
...

If there are two generation processes, the generation proceeds in parallel as follows:

Process 0 of 2:  
Process 1 of 2:
RF(seed,0)  
RF(seed,100)  
Generate rows 0-99  
Generate rows 100-199
RF(seed,200)  
RF(seed,300)  
Generate rows 200-299  
Generate rows 300-399
...
...

The random number generator is always re-seeded to a deterministic value (based on the user-specified seed and the row number) before generating a swath. Thus any particular swath will be generated identically regardless of the number of processes participating in the generation.

Note that the load balancing for Algorithm 1 is even; each generation process generates at most $SWATHSIZE$ more rows than any other generation process.

As an example, consider the following SDDL table definition:

```xml
<database>
  <seed>1367281</seed>
  ...
  <table name="points" length="10">
    <field name="X" type="real">
      <min>0.0</min>
      <max>100.0</max>
    </field>
    <field name="Y" type="real">
      ...
```
\[ X \times 1.5 - 5.0 \]

The resulting relation would contain ten rows of \( X \) and \( Y \) attributes. Each \( X \) value would be between 0 and 100 (inclusive). Each \((X,Y)\) pair would be a point along the line \( Y = 1.5 \times X - 5.0 \).

Assuming a SWATHSIZE of 3, a single-process generation would take the following actions (the actual data is hypothetical for a SWATHSIZE of 3):

Process 0 of 1:
RF(1367281, 0) Generate row 0 = (24.28, 31.42) Generate row 1 = (91.18, 131.77) Generate row 2 = (96.66, 139.99) RF(1367281, 3) Generate row 3 = (15.85, 18.775) Generate row 4 = (83.97, 120.955) Generate row 5 = (67.44, 96.16) RF(1367281, 6) Generate row 6 = (4.04, 1.06) Generate row 7 = (3.6, 0.4) Generate row 8 = (80.0, 115.0) RF(1367281, 9) Generate row 9 = (21.95, 27.925)

Again assuming a SWATHSIZE of 3, generation over two processes would involve the following actions:

Process 0 of 2:
RF(1367281, 0) Generate row 0 = (24.28, 31.42) Generate row 1 = (91.18, 131.77) Generate row 2 = (96.66, 139.99) RF(1367281, 6) Generate row 6 = (4.04, 1.06) Generate row 7 = (3.6, 0.4) Generate row 8 = (80.0, 115.0)
Process 1 of 2:
RF(1367281, 3)
Generate row 3 = (15.85,18.775)
Generate row 4 = (83.97,120.955)
Generate row 5 = (67.44,96.16)
RF(1367281, 9)
Generate row 9 = (21.95,27.925)

In this fashion, the swaths are handled in a round-robin fashion by the participating generation processes. In terms of load balancing, Process 0 generated six rows and Process 1 generated four rows, which puts them within SWATHSIZE (3) rows of each other.

4.1.2 Generation Algorithm 2

When an iteration constraint is present in a table, different rules must be used for partitioning the table and generating it in parallel. When iterations are present, it becomes more difficult to deal out constant-sized swaths to each generation process.

Consider the following SDDL table definition:

<table name="iteration_example">
  <field name="deptID" type="int">
    <iteration query="SELECT ID FROM departments"/>
  </field>
  <field name="courseID" type="int">
    <iteration query="SELECT ID
        FROM courses
        WHERE deptID = [deptID]"/>
  </field>
</table>

How many courseID rows will be generated for each deptID value? One can’t really tell before running the actual queries. As a consequence, one doesn’t know a priori how to divide the output into equal-sized slices.
Furthermore, it would be inefficient to have all generation processes run all possible queries during data generation. In the example above, suppose that there were 100 unique ID values in the departments table being queried. That means that 101 queries must be performed in order to generate the data (1 to the departments table, 100 to the courses table). Now suppose that there were 10 processes participating in the generation of data. Would we want all 10 processes submitting 101 queries? This sort of behavior would prevent the linear scalability that is being sought through parallel data generation. (While query iterations have been used here as an example, the same potential for serial behavior exists with nested count iterations and pool iterations as well.)

Instead, when iterations are present in a table, each outer iteration element (OIE) is considered a swath”, and the data is partitioned on outer iteration elements. Each generation process knows the number of elements in any outer iteration. For query iterations, it is the number of elements returned by the given query. For pool iterations, it is the number of choices in the specified pool. For numeric iterations, it is the value of the count attribute. Therefore, the table is sliced up into outer iteration elements.

In general, generation process $P$ of $N$ will take the following action during generation algorithm 2:

```
Process P of N:
RF(seed, P)  # Generate rows for OIE P
RF(seed, P+N)  # Generate rows for OIE (P+N)
RF(seed, P+2N)  # Generate rows for OIE (P+2N)
...
```
When a single generation process performs generation algorithm 2, the following occurs:

Process 0 of 1:
RF(seed,0)
Generate OIE 0
RF(seed,1)
Generate OIE 1
RF(seed,2)
Generate OIE 2
...

With 2 processes, the generation proceeds as follows:

Process 0 of 2:  Process 1 of 2:
RF(seed,0)  RF(seed,1)
Generate OIE 0  Generate OIE 1
RF(seed,2)  RF(seed,3)
Generate OIE 2  Generate OIE 3
...
...

Again, the random number generator is always re-seeded to a deterministic value before generating the rows associated with an outer iteration element. Thus the rows associated with any OIE will be generated identically regardless of the number of processes participating in the generation.

Note that generation algorithm 2 is balanced with respect to the number of OIEs assigned to the generation processes; each generation process will handle the generation of at most one more OIE than its peers. However, the number of rows associated with an OIE is not guaranteed to be constant. Therefore, it is possible for the generation to be unbalanced in terms of the number of rows generated per generation process.

For an example of generation algorithm 2, consider the following SDDL table definition:

```xml
<database>
  <seed>43291756</seed>
</database>
```
<pool name="colors">
   <choice name="red"/>
   <choice name="orange"/>
   <choice name="yellow"/>
   <choice name="green"/>
   <choice name="blue"/>
   <choice name="violet"/>
</pool>

<table name="items">
   <field name="color" type="CHAR(6)">
      <iteration pool="colors"/>
   </field>
   <field name="ID" type="int">
      <iteration query="SELECT ID FROM products WHERE color='[color]'"/>
   </field>
</table>
</database>

Suppose also that the “products” table contained the following \{ID, color\} tuples:

\{1,'blue'},\{2,'green'},\{3,'yellow'},\{4,'red'},
\{5,'green'},\{6,'violet'},\{7,'orange'},\{8,'orange'},
\{9,'yellow'},\{10,'green'},\{11,'violet'},\{12,'blue'\}.

If the table above were to be generated using a single generation process, generation would proceed as follows:

Process 0 of 1:
RF(43291756, 0), run query for ‘red’
Generate row (‘red’,4)
RF(43291756, 1), run query for ‘orange’
Generate row (‘orange’,7)
Generate row (‘orange’,8)
RF(43291756, 2), run query for ‘yellow’
Generate row (‘yellow’,3)
Generate row (‘yellow’,9)
RF(43291756, 3), run query for ‘green’
Generate row (‘green’,2)
Generate row (‘green’,5)
Generate row (‘green’,10)
RF(43291756, 4), run query for ‘blue’
Generate row (‘blue’,1)
Generate row (‘blue’,12)
RF(43291756, 5), run query for ‘violet’
Generate row (‘violet’,6)
Generate row (‘violet’,11)

If 2 processes were used to generate the table, the generation would proceed as follows:

Process 0 of 2:
RF(43291756, 0), run query for ‘red’
Generate row (‘red’,4)
RF(43291756, 2), run query for ‘yellow’
Generate row (‘yellow’,3)
Generate row (‘yellow’,9)
RF(43291756, 4), run query for ‘blue’
Generate row (‘blue’,1)
Generate row (‘blue’,12)

Process 1 of 2:
RF(43291756, 1), run query for ‘orange’
Generate row (‘orange’,7)
Generate row (‘orange’,8)
RF(43291756, 3), run query for ‘green’
Generate row (‘green’,2)
Generate row (‘green’,5)
Generate row (‘green’,10)
RF(43291756, 5), run query for ‘violet’
Generate row (‘violet’,6)
Generate row (‘violet’,11)

Note the following from the parallel generation sequence shown above:

- Not only is the data partitioned, but the queries are divided between the generation processes as well.

- Generation algorithm 2 can result in an unbalanced load. Although each process handled 3 OIEs, process 0 generated 5 rows while process 1 generated 7 rows. In general, this imbalance is minimized when generating a large data set with a large number of OIEs.
4.1.3 Generation Algorithm 3

For the sake of completeness, the rarely used “punting” algorithm (generation algorithm 3) will also be discussed. Generation algorithm 3 is used to generate table data for tables that contain both iterations and auto-incremented integer fields. In this case, generation algorithm 1 is not optimal due to the presence of an iteration. Generation algorithm 2 is not useful because it is not possible to produce continuous values for the auto-incremented field without every processor performing the inner iterations (which would be inefficient).

Instead, generation algorithm 3 will generate all data from generation process 0. In effect, generation algorithm 3 enforces the serial generation of the data by inhibiting all generation processes with an index greater than 0 from participating in the generation of data.

Consider the following SDDL table definition:

```xml
<database>
<table name="Courses">
  <field name="CourseID" type="int">
    <autoIncrement/>
    <min>1000</min>
  </field>
  <field name="Dept" type="CHAR(8)">
    <iteration query="SELECT DeptName
                     FROM departments"/>
  </field>
  <field name="CourseName" type="CHAR(48)">
    <iteration query="SELECT CourseName
                     FROM catalog
                     WHERE DeptName = '[Dept]'"/>
  </field>
</table>
</database>
```
The intent here is to generate a table of courses and their associated departments, assigning a monotonically increasing CourseID to each course generated. The table definition above contains both an auto-incremented field CourseID and iteration constrained fields Dept and CourseName. Hence, the Courses table would be generated via generation algorithm 3, and only process 0 would participate in the generation.

4.2 Iteration Variables and Parallelism

Some data sets require very tight inter-row relationship constraints during generation. For example, suppose you wanted to track and timestamp the progress of a number of trucks travelling between cities on a map. If you want to know, “where is truck A now?”, you must answer the question, “where was truck A previously?”. If you want to know, “At what time did truck B arrive at location C?”, you must answer the question(s), “Where is truck B coming from, and what was the timestamp at that location?”. Thus, the generation of one row is tightly dependent upon values from previously generated rows.

The maintenance of any sort of global variable set across parallel processes would be an inter-process communications nightmare. Therefore, it is not desirable to support shared variables in the data generator. How, then, can tight inter-row relationships be supported?

The answer is a mechanism called an iteration variable. Iteration variables allow values to be tracked within an iteration element. Since an iteration element will never be split across generation processes (according to generation algorithm 2, described earlier
in this chapter), there is no need to communicate the values of iteration variables across processes.

Consider the following SDDL pool definition:

```xml
<pool name="map">
  <choice name="LA">
    <pool name="neighbors">
      <choice name="Phoenix">
        <miles>390</miles>
      </choice>
      <choice name="Salt Lake City">
        <miles>690</miles>
      </choice>
      <choice name="Seattle">
        <miles>1130</miles>
      </choice>
    </pool>
  </choice>
  <choice name="Phoenix">
    <pool name="neighbors">
      <choice name="LA">
        <miles>390</miles>
      </choice>
      <choice name="Salt Lake City">
        <miles>650</miles>
      </choice>
      <choice name="Seattle">
        <miles>850</miles>
      </choice>
    </pool>
  </choice>
  <choice name="Salt Lake City">
    <pool name="neighbors">
      <choice name="Phoenix">
        <miles>650</miles>
      </choice>
      <choice name="LA">
        <miles>690</miles>
      </choice>
      <choice name="Seattle">
        <miles>850</miles>
      </choice>
    </pool>
  </choice>
</pool>
```

4 Distances obtained from http://www.travelnotes.org/NorthAmerica/distances.htm
The map pool above models the western United States (simplistically) as follows:

![Figure 4-2: Map modeled by "map" Pool](image)

How can timestamp/location information be generated for synthetic trucks traversing this map? It could be done with the following SDDL table definition (which references the map pool above):

1. `<table name="trucks">
2.  `<field name="truckID" type="int">
3.   `<iteration base="1" count="5">
4.     `<repeatMin>3</repeatMin>
5.     `<repeatMax>5</repeatMax>
6.   `<itervar name="currCity"..."/>
This table definition will produce map traversal data for five trucks, with ids 1 through 5 (line 3). The truckID iteration will produce anywhere from 3 to 5 waypoints for each truck (via the repeatMin/repeatMax elements in lines 4-5). Each iteration element (i.e., each truck) will have two iteration variables: One to keep track of the current city (line 6), and one to keep track of the current time (line 7). The currCity variable will be initialized to a city in the map pool at the start of each iteration element; the currTime variable will be initialized to noon on July 12, 2006, at the start of each iteration element. Thus, each truck will begin its simulated existence at one of the map locations on noon, July 12, 2006.
The city field (lines 10-12) will simply output the value of the currCity iteration variable. The tstamp field (lines 13-15) will output the value of the currTime iteration variable.

The newCity and newTime variables (lines 16-21) handle the logic of updating the currCity and currTime iteration variables, respectively. In line 17, the currCity variable is changed to one of the neighboring cities of the current city. In line 20, the currTime variable is incremented by the number of seconds that it takes to drive from the current city (city) to the next city (currCity), assuming that the truck is driving 55 miles per hour.

The actions employed by a single generation process would look like this:

Process 0 of 1:
RF(seed, 0)
initialize currCity to random city
initialize currTime to 2006-07-12 12:00:00
Generate OIE 0 (truck 0)
RF(seed, 1)
initialize currCity to random city
initialize currTime to 2006-07-12 12:00:00
Generate OIE 1 (truck 1)
RF(seed, 2),
initialize currCity to random city
initialize currTime to 2006-07-12 12:00:00
Generate OIE 2 (truck 2)
RF(seed, 3),
initialize currCity to random city
initialize currTime to 2006-07-12 12:00:00
Generate OIE 3 (truck 3)
RF(seed, 4),
initialize currCity to random city
initialize currTime to 2006-07-12 12:00:00
Generate OIE 4 (truck 4)

The actions employed by dual generation processes would look like this:

Process 0 of 2:
RF(seed, 0)
initialize currCity to random city
initialize currTime to 2006-07-12 12:00:00
Generate OIE 0 (truck 0)
RF(seed, 2),
initialize currCity to random city
initialize currTime to 2006-07-12 12:00:00
Generate OIE 2 (truck 2)
RF(seed, 4),
initialize currCity to random city
initialize currTime to 2006-07-12 12:00:00
Generate OIE 4 (truck 4)

Process 1 of 2:
RF(seed, 1)
initialize currCity to random city
initialize currTime to 2006-07-12 12:00:00
Generate OIE 1 (truck 1)
RF(seed, 3),
initialize currCity to random city
initialize currTime to 2006-07-12 12:00:00
Generate OIE 3 (truck 3)

Because the iteration variables are initialized after the reseed function is called, they are guaranteed to be initialized identically for any outer iteration element regardless of the number of processes employed during data generation. And since subsequent data within an iteration element depends upon the initialized values of the iteration variables, the data for the iteration element is guaranteed to be generated identically regardless of the degree of parallelism employed during generation.

To verify this, here is the output from generating the trucks table using a single generation process:

<table>
<thead>
<tr>
<th>truckID</th>
<th>city</th>
<th>tstamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'Seattle'</td>
<td>'2006-07-12 12:00:00'</td>
</tr>
<tr>
<td>1</td>
<td>'Salt Lake City'</td>
<td>'2006-07-13 03:27:16'</td>
</tr>
<tr>
<td>1</td>
<td>'LA'</td>
<td>'2006-07-13 15:59:59'</td>
</tr>
<tr>
<td>2</td>
<td>'Seattle'</td>
<td>'2006-07-12 12:00:00'</td>
</tr>
<tr>
<td>2</td>
<td>'LA'</td>
<td>'2006-07-13 08:32:43'</td>
</tr>
<tr>
<td>2</td>
<td>'Phoenix'</td>
<td>'2006-07-13 15:38:10'</td>
</tr>
<tr>
<td>3</td>
<td>'LA'</td>
<td>'2006-07-12 12:00:00'</td>
</tr>
</tbody>
</table>
Table 4-1: Trucks Table Produced by Single Process

When the data is generated over two processes, the following tables are produced:

<table>
<thead>
<tr>
<th>truckID</th>
<th>city</th>
<th>tstamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'Seattle'</td>
<td>'2006-07-12 12:00:00'</td>
</tr>
<tr>
<td>1</td>
<td>'Salt Lake City'</td>
<td>'2006-07-13 03:27:16'</td>
</tr>
<tr>
<td>3</td>
<td>'LA'</td>
<td>'2006-07-12 15:59:59'</td>
</tr>
<tr>
<td>3</td>
<td>'LA'</td>
<td>'2006-07-13 12:00:00'</td>
</tr>
<tr>
<td>3</td>
<td>'Phoenix'</td>
<td>'2006-07-12 19:05:27'</td>
</tr>
<tr>
<td>3</td>
<td>'LA'</td>
<td>'2006-07-13 02:10:54'</td>
</tr>
<tr>
<td>3</td>
<td>'Salt Lake City'</td>
<td>'2006-07-13 14:43:37'</td>
</tr>
<tr>
<td>5</td>
<td>'Salt Lake City'</td>
<td>'2006-07-12 12:00:00'</td>
</tr>
<tr>
<td>5</td>
<td>'LA'</td>
<td>'2006-07-13 00:32:43'</td>
</tr>
<tr>
<td>5</td>
<td>'Seattle'</td>
<td>'2006-07-13 21:05:26'</td>
</tr>
<tr>
<td>5</td>
<td>'LA'</td>
<td>'2006-07-14 17:38:09'</td>
</tr>
</tbody>
</table>

Table 4-2: Trucks Table Output from Process 0 of 2

<table>
<thead>
<tr>
<th>truckID</th>
<th>city</th>
<th>tstamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>'Seattle'</td>
<td>'2006-07-12 12:00:00'</td>
</tr>
<tr>
<td>2</td>
<td>'LA'</td>
<td>'2006-07-13 08:32:43'</td>
</tr>
<tr>
<td>2</td>
<td>'Phoenix'</td>
<td>'2006-07-13 15:38:10'</td>
</tr>
<tr>
<td>4</td>
<td>'LA'</td>
<td>'2006-07-12 12:00:00'</td>
</tr>
<tr>
<td>4</td>
<td>'Phoenix'</td>
<td>'2006-07-12 19:05:27'</td>
</tr>
<tr>
<td>4</td>
<td>'LA'</td>
<td>'2006-07-13 02:10:54'</td>
</tr>
<tr>
<td>4</td>
<td>'Phoenix'</td>
<td>'2006-07-13 09:16:21'</td>
</tr>
<tr>
<td>4</td>
<td>'LA'</td>
<td>'2006-07-13 16:21:48'</td>
</tr>
</tbody>
</table>

Table 4-3: Trucks Table Output from Process 1 of 2
This is a simple example of generating data that exhibits tight inter-row dependencies. There are a number of ways in which complexity could be added to the model: more cities on the map, variable speeds for the trucks, factoring in a random rest time in each city, and adding more trucks, for example. However, this simple example shows that SDG/SDDL can support tight inter-row dependencies (using iteration variables) without compromising the ability to produce deterministic data in parallel.
5 DESCRIPTIVE ADEQUACY OF SDDL

The claim has been made in this document that SDDL provides powerful mechanisms for describing the characteristics of data to be generated. This chapter attempts to answer the question, “how descriptive is SDDL?” This is done by demonstrating the manner in which SDDL can be used to enforce some of the more commonly used constraint types found in the relational and E-R data models.

The constraint types that will be featured in this chapter include the following:

- Domain Constraints
- Key Constraints
- Entity Integrity Constraints
- Referential Integrity Constraints
- Cardinality Constraints

While this is certainly not an exhaustive list of constraint types or data modeling concepts, it is a list of the more common constraint types. Being able to enforce these constraint types gives SDDL a sufficient “critical mass” of capabilities to be able to describe most real-world situations.

The SQL examples given in this chapter are all in SQLPlus, Oracle’s SQL dialect. The examples herein were derived from examples in [21].

5.1 Domain Constraints

In the relational model, within each tuple, the value for each attribute must be a value from that attribute’s domain. This is called a domain constraint ([22], p. 154).
Domain constraints can be sub-divided into two classes: type constraints and value constraints.

5.1.1 Type Constraints

In SQL, every attribute in a table is assigned a domain type when the table is created. For example, consider the following SQL table creation statement:

```sql
CREATE TABLE Employees (ID INTEGER, L_Name CHAR(10), F_Name CHAR(15))
```

The result of the statement above is that a table named Employee is created. The Employee table has three attributes: ID, L_Name, and F_Name. The ID attribute has a domain type constraint of INTEGER, and the L_Name and F_Name attributes have domain type constraints of CHAR(10) and CHAR(15), respectively.

In SDDL, each field element has a type attribute. SDDL type attributes can be any of these common SQL attribute types: BOOLEAN, CHAR(n), VARCHAR(n), NUMERIC(m,n), DECIMAL(m,n), DOUBLE, INTEGER, DATE, TIME, and TIMESTAMP. The CREATE TABLE statement above can be directly converted into SDDL as follows:

```xml
<table name="Employees" length="<<some_length>>">
  <field name="ID" type="int">
    <<value constraint for ID>>
  </field>
  <field name="L_Name" type="CHAR(10)">
    <<value constraint for L_Name>>
  </field>
  <field name="F_Name" type="CHAR(15)">
    <<value constraint for F_Name>>
  </field>
</table>
```
While there are some SQL domain types not currently supported by SDDL (such as BLOB, CLOB and LONGINT), SDDL provides coverage of the common domain types supported by SQL.

5.1.2 Value Constraints

Some type constraints can technically be considered as value constraints. For example, an INTEGER attribute constrains its values to be in the range of 32-bit integers. However, type constraints are very weak value constraints.

The two most common SQL value constraint types\(^5\) are range constraints and discrete set constraints, as illustrated in the following CREATE TABLE statement:

```sql
CREATE TABLE Students (Name CHAR(32), ID INTEGER,
Class CHAR(2) CHECK(Class in ‘FR’,’SO’,’JR’,’SR’),
GPA NUMERIC(3,2) CHECK(GPA >= 0.0 AND GPA <=4.0))
```

In the Students table created above, the Class attribute must come from the value set \{ ‘FR’, ‘SO’, ‘JR’, ‘SR’ \}. The GPA attribute must be between 0.0 and 4.0, inclusive. We could enforce these value constraints in SDDL as follows:

```xml
<pool name="classes">
  <choice name="FR"/>
  <choice name="SO"/>
  <choice name="JR"/>
  <choice name="SR"/>
</pool>

<table name="Students" length="<<some_length>>">
  <field name="Name" type="CHAR(32)">
```

\(^5\) Of course, “NOT NULL” value constraints are also very common. These can be enforced in SDDL by simply not allowing for the output of a NULL value for a field.
In the SDDL table definition above, the value for the Class field is defined to be one of the entries from the classes pool. The value for the GPA field is defined to be a random number between 0.00 and 4.00.

In this manner, SDDL is able to enforce the value constraints supported by SQL.

It is worth noting that SDDL allows for descriptive constraints beyond those provided by SQL. For example, SDDL can specify that “I want 40% of these values to be between 50 and 60, and 60% of the values to be between 80 and 100”, or “I want the mean of the values to be 65.2 and the standard deviation to be 5.7”.

### 5.2 Key Constraints

In the relational data model, each value of a key attribute must be unique. This is called a key constraint ([22], p. 66). Key constraints apply to multi-attribute keys as well as single-attribute keys.
5.2.1 Single-Attribute Keys

For a single-attribute key, enforcing the key constraint is relatively simple, especially for numeric keys. For instance, suppose you wanted to generate 100 records, each of which had a unique numeric ID field. The ID field could be described in SDDL as follows:

```xml
<field name="ID" type="int">
    <iteration base="1" count="100"/>
</field>
```

The ID field generated by the above code will generate the value 1 through 100, with no repeats. Thus, if ID is the key for the relation, then the key constraint is preserved (in this case).

It is important to use SDDL iterations when generating key values, and that the iteration elements have a maximum repeat count of 1 (which is the default). The use of iterations in such fashion is the only way to guarantee the uniqueness of the values being generated. Consider what would happen if you used a queryPool to generate a key value:

```xml
<field name="ID" type="int">
    <queryPool query="SELECT ID FROM Students"/>
</field>
```

The ID values generated from the description above are not guaranteed to be unique. Neither are those generated by the following description:

```xml
<field name="ID" type="int">
    <min>1</min>
    <max>100</max>
</field>
```
What about non-numeric keys? Suppose that our key were a field called color, the domain of which was the values \{red, green, blue\}. In order to make sure that each key was unique, we could simply iterate through all possible key values:

```
<pool name="colors">
  <choice name="red"/>
  <choice name="green"/>
  <choice name="blue"/>
</pool>
```

...  

```
<field name="color" type="CHAR(5)">
  <iteration pool="colors"/>
</field>
```

The color field would contain the values red, green and blue, with no repeats.

It is also possible to generate unique values without exhausting the value set in question. Techniques for doing this will be described in the next subsection.

5.2.2 Multi-Attribute Keys

One can enforce key constraints for multi-attribute keys in a similar manner to that described for single-attribute keys. Suppose that you already had two tables: a courses table with information about courses being taught at an institution, and a students table with information about each student enrolled in that institution. Now suppose that you wanted to create a roster table to track which students were enrolled in which classes. The only two attributes for the roster table would be CourseID and StudentID, and together they would comprise the primary key for that table.
The roster table might be modeled in SDDL as follows:

```xml
<table name="roster">
  <field name="StudentID" type="int">
    <iteration query="SELECT ID FROM Students" />
  </field>
  <field name="CourseID" type="int">
    <iteration query="SELECT ID FROM Courses" />
  </field>
</table>
```

The roster table defined above certainly works, but it enrolls every student in every course. Can we make it more realistic, without violating the key constraint? The answer is “yes”. We need to keep the iteration structure intact to make sure that we do not repeat primary key values. But we can constrain one or both iterations to “skip” elements, and we can constrain students to only take courses in their own departments as follows:

```xml
<table name="roster">
  <field name="StudentID" type="int">
    <iteration query="SELECT ID,Dept FROM Students" >
      <repeatDist>
        <tier prob="0.9" min="1" max="1"/>
        <tier prob="0.1" min="0" max="0"/>
      </repeatDist>
    </iteration>
  </field>
  <field name="CourseID" type="int">
    <iteration query="SELECT ID FROM Courses where Dept = [StudentID$Dept]" >
      <repeatDist>
        <tier prob="0.25" min="1" max="1"/>
        <tier prob="0.75" min="0" max="0"/>
      </repeatDist>
    </iteration>
  </field>
</table>
```
In the roster table defined above, roughly 10% of the students will not be registered for any class, and each student registered will register for about 25% of the courses being offered in the student’s department. In this fashion, it is possible to enforce key constraints without exhausting all possible values for the key.

5.2.3 Entity Integrity Constraints

In the relational model, a primary key value cannot be NULL. If the primary key is composed of multiple attributes, then none of the participating attributes are allowed to have a value of NULL. This is called an entity integrity constraint ([22], p. 157) because each row in the table references an entity instance that is being modeled in the database.

In SDDL, NULL attribute values are never produced by default. Instead, the SDDL user must specify the instances where NULLs should be output. To enforce an entity integrity constraint is a matter of not specifying that NULL values are to be output for a primary key attribute.

5.3 Referential Integrity Constraints

In the relational model, a foreign key value, if not null, must be present in the primary key of the table that the foreign key references. This is called a referential integrity constraint ([22], p. 158).

In general, referential integrity constraints can be enforced in three ways using SDDL:

- Query iterations. A query iteration will loop through all possible valid values of a foreign key. One would typically use a query iteration to
model a referential integrity constraint when one desires the foreign key to be a unique key as well.

- Query pools. A query pool will choose one of all possible valid values of a foreign key each time that a row is generated. This method allows for the repeating of foreign key values in the referencing table.

- Continuous range generation. If one knows *a priori* that the values of a foreign key are constrained to a specific continuous integer range, for example, then either a count iteration or a min/max constraint can be used to generate valid foreign key values. The count iteration could be used to generate unique values for the foreign key, while the min/max constraint would allow for repeat foreign key values. Typically, foreign key values being bunched into continuous integer ranges only occurs in contrived data definitions.

Consider the following CREATE TABLE statement:

```sql
CREATE TABLE Sales (
    ItemID INTEGER,
    StoreID INTEGER,
    SalePrice DECIMAL(9,2),
    FOREIGN KEY (ItemID) REFERENCES Items(ID),
    FOREIGN KEY (StoreID) REFERENCES Stores(ID))
```

The table created above would have three attributes: `ItemID`, `StoreID` and `SalePrice`. `ItemID` and `StoreID` are foreign keys, because they reference primary keys from other tables. `ItemID` and `StoreID` are constrained to be actual values from the ID attributes of the `Items` and `Stores` tables, respectively.
These referential integrity constraints could be enforced in SDDL with query pools as follows:

```xml
<table name="Sales" length="<<some_length>>">
  <field name="ItemID" type="int">
    <queryPool query="SELECT ID FROM Items"/>
  </field>
  <field name="StoreID" type="int">
    <queryPool query="SELECT ID FROM Stores"/>
  </field>
  <field name="SalePrice" type="DECIMAL(9,2)">
    <<value description for SalePrice>>
  </field>
</table>
```

The same referential integrity constraints could also be enforced with query iterations as follows:

```xml
<table name="Sales">
  <field name="ItemID" type="int">
    <iteration query="SELECT ID FROM Items"/>
  </field>
  <field name="StoreID" type="int">
    <iteration query="SELECT ID FROM Stores"/>
  </field>
  <field name="SalePrice" type="DECIMAL(9,2)">
    <<value description for SalePrice>>
  </field>
</table>
```

The difference between the two methods of referential integrity enforcement is that the query pool method results in a fixed length table and allows foreign key values to repeat, and the query iteration method exhaustively produces all valid combinations of ItemID and StoreID without repeating any combinations.

### 5.4 Cardinality Constraints

In E-R diagrams, one can specify minimum and maximum participation counts for each record of each table. For example, consider the following E-R diagram:
In Figure 5-1, each entity in the Salesman table references between \( s_{min} \) and \( s_{max} \) (inclusive) entities in the Region table. Each entity in the Region table is associated with between \( r_{min} \) and \( r_{max} \) (inclusive) entities in the Salesman table.

These relational concepts are associated with the **minimum** cardinality of an entity:

- If an entity has a minimum cardinality of 1 (or more), we say that the entity *fully (or totally) participates* ([22], p. 75) in a relationship. For example, suppose that each salesman in Figure 5-1 is required to have an associated region; \( s_{min} \) would then be 1 or greater, and we would say that the Salesman entity fully participates in the "Sells In" relationship.

- If an entity has a minimum cardinality of 0, we say that the entity *partially participates* ([22], p. 75) in a relationship. If, in Figure 5-1, a region was not required to have any salesmen, then \( r_{min} \) would be 0 and Region would partially participate in the “Sells In” relationship.
Note that it is possible for a partially participating entity to have a fully participated instance, but it is not possible for a fully participating entity to have a partially participated instance.

These relational concepts are associated with the maximum cardinalities of the entities involved in a binary relationship:

- If the maximum cardinality on both sides is one, then the relationship is a *one-to-one (or 1:1) relationship* ([22], p. 74). In our example, if there could be no more than one salesman per region, and a salesman could be assigned to only one region, then $s_{max}=r_{max}=1$ and “Sells In” would be a one-to-one relationship.

- If the maximum cardinality on one side is one, and on the other side is more than one, then we call the relationship a *one-to-many (or 1:N) relationship* ([22], p. 74). In our example, if a salesman could be assigned to only one region, but a region could be assigned to multiple salesmen, then $s_{max}=1$ and $r_{max}=(N>1)$, and “Sells In” would be a one-to-many relationship.

- If the maximum cardinality on each side is more than one, then the relationship is considered a *many-to-many (or M:N) relationship* ([22], p. 74). Going back to our example, if a salesman could be assigned to multiple regions, and a region could contain multiple salesmen, then $s_{max}=(M>1)$ and $r_{max}=(N>1)$, and “Sells In” would be a many-to-many relationship.
It is possible in SDDL to enforce full or partial participation in a relationship, as well as one-to-one, one-to-many, and many-to-many relationships. All combinations are covered in the ensuing subsections. While the SDDL code given as solutions to the various cardinality combinations to follow will reference our Salesman/Region example, the techniques employed could be applied to generating any table that is to be related to a pre-existing table. The point of these examples is to demonstrate that SDDL is capable of enforcing cardinality constraints.

Note that all relationships in sections 5.4.1, 5.4.2 and 5.4.3 are assumed to be binary relationships; general n-ary relationships will be discussed section 5.4.4. It is also assumed that the table on the left-hand side (“LHS”) of the relationship is being generated, while the table on the right-hand side (“RHS”) of the relationship already exists. In all examples, the Region table is assumed to have been previously generated as follows:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>North</td>
</tr>
<tr>
<td>2</td>
<td>South</td>
</tr>
<tr>
<td>3</td>
<td>East</td>
</tr>
<tr>
<td>4</td>
<td>West</td>
</tr>
</tbody>
</table>

Table 5-1: Assumed Contents of Region Table

### 5.4.1 One-to-One Relationships

In a one-to-one relationship, one or the other of the tables involved should contain a foreign key linking the two tables together. From our example in Figure 5-1, if “Sells

---

6 A useful extension of SDG would be the capability to automatically convert an E-R diagram into SDDL based on these techniques. Such a possibility is mentioned in Section 12.3 on Future Directions.
In a one-to-one relationship, then the schemas for the *Salesman* and *Region* tables could look like this:

**Salesman**

| ID: INTEGER | Name: CHAR(16) | RegionID: INTEGER |

Table 5-2: Layout of Salesman Table for 1-1 Relationship

**Region**

| ID: INTEGER | Name: CHAR(5) |

Table 5-3: Layout of Region Table for 1-1 Relationship

Assuming the *Region* data was already generated as in Table 5-1, one would want to now generate the *Salesman* table. This could be done in several ways, depending upon the desired participation level of the *Salesman* and *Region* tables.

5.4.1.1 LHS partially participating, RHS partially participating

- LHS cardinality constraint: 0..1
- RHS cardinality constraint: 0..1

In this situation, our example *Salesman* table could be defined in SDDL as follows:

```xml
database
  <seed>7829683417</seed>
  <import filename="names.xml"/>
  <table name="Salesman">
    <variable name="rID" type="int">
      <iteration query="SELECT ID FROM Region"/>
    </variable>
    <field name="ID" type="int">
      <formula>rID*5</formula>
    </field>
    <field name="Name" type="CHAR(16)">
```

124
For the RegionID attribute, there is a chance of generating a NULL, which would mean that the associated record would not be participating in the relationship. Also, there is no guarantee that all of the records in the Region table will participate in the “Sells In” relationship. Each entity on each side is guaranteed to participate no more than one time in the relationship.

The table generated from the SDDL definition above looks like this:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>RegionID</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Joseph</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Anthony</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>Joshua</td>
<td>NULL</td>
</tr>
<tr>
<td>20</td>
<td>Nicholas</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Table 5-4: Salesman Table with $smin=0, smax=1, rmin=0, rmax=1$

Table 5-4 indicates that IDs 15 and 20 from the Salesman table did not participate in the “Sells In” relationship. Also, IDs 3 and 4 from the Region table did not participate in the “Sells In” relationship. Finally, no entity from either side

---

\[7\] The imported names.xml file contains a names pool, which is sorted into a number of different ethnic groups. For simplicity, all examples in this chapter choose a name from the Caucasian ethnic group.
participates more than once in the relationship. Thus, the SDDL table definition given above allows for partial participation from each entity in a 1-1 relationship.

5.4.1.2 LHS fully participating, RHS partially participating

- LHS cardinality constraint: 1..1
- RHS cardinality constraint: 0..1

In this situation, using our example Salesman/Region tables, we could model the Salesman table in SDDL as follows:

```xml
<database>
  <seed>7829683417</seed>
  <import filename="names.xml"/>
  <table name="Salesman">
    <variable name="rID" type="int">
      <iteration query="SELECT ID FROM Region">
        <repeatMin>0</repeatMin>
        <repeatMax>1</repeatMax>
      </iteration>
    </variable>
    <field name="ID" type="int">
      <formula>rID*100</formula>
    </field>
    <field name="Name" type="CHAR(16)">
      <formula>names['Caucasian'].firstnames</formula>
    </field>
    <field name="RegionID" type="int">
      <formula>rID</formula>
    </field>
  </table>
</database>
```

In the SDDL definition above, the rID iteration insures that we select each element of the Region table 0 or 1 times. For each of these selections, we create a corresponding salesman. The generated table looks like this:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>RegionID</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Olivia</td>
<td>2</td>
</tr>
</tbody>
</table>

126
Table 5-5: Salesman Table with $s_{min}=1$, $s_{max}=1$, $r_{min}=0$, $r_{max}=1$

Table 5-5, while simple, fulfills the requirements. Each entity in the Salesman table participated in the relationship, but only one of four entries from the Region table participated. Also, no entity from either table participated more than once in the relationship.

5.4.1.3 LHS partially participating, RHS fully participating

- LHS cardinality constraint: 0..1
- RHS cardinality constraint: 1..1

In this situation, using our Salesman/Region example, the Salesman table can be described in SDDL as follows:

```xml
<database>
  <seed>7829683417</seed>
  <import filename="names.xml"/>
  <table name="Salesman">
    <variable name="rID" type="int">
      <iteration query="SELECT ID FROM Region">
        <itervar name="counter" type="int" init="0"/>
        <repeatDist>
          <tier prob="0.5" min="1" max="1"/>
          <tier prob="0.5" min="2" max="3"/>
        </repeatDist>
      </iteration>
    </variable>
    <field name="ID" type="int">
      <formula>rID*5+counter</formula>
    </field>
    <field name="Name" type="CHAR(16)">
      <formula>names['Caucasian'].firstnames</formula>
    </field>
    <field name="RegionID" type="int">
      <formula>(counter==0)?rID:"NULL"</formula>
    </field>
    <variable name="incCounter" type="int">
      <formula>counter=counter+1</formula>
    </variable>
  </table>
</database>
```
The rID iteration assures that each ID from the Region table is visited at least once, and possibly 2 or 3 times. However, the counter iteration variable is used to assure that each RegionID will only be used once; on the second or third repetition of that iteration element, “NULL” will be output. These “NULL” values also serve to model partial participation from the Salesman table. The Salesman table from this definition is output as follows:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>RegionID</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>James</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Jessica</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>James</td>
<td>NULL</td>
</tr>
<tr>
<td>15</td>
<td>Zachary</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>Emily</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5-6: Salesman Table with smin=0, smax=1, rmin=1, rmax=1

Indeed, each Region entry participates exactly once in the relationship, and each Salesman entity participates 0 or 1 times in the relationship.

5.4.1.4 LHS fully participating, RHS fully participating

- LHS cardinality constraint: 1..1
- RHS cardinality constraint: 1..1

This is one of the easier situations to model. By definition, if two tables both fully participate in a one-to-one relationship, then the tables must have equal cardinality.

Using our Salesman/Region example, the Salesman table can be modeled in SDDL as follows:
In the table definition above, a Salesman entry is generated for each entry in the Region table. The table is generated as follows:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>RegionID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Joseph</td>
<td>1</td>
</tr>
<tr>
<td>200</td>
<td>Anthony</td>
<td>2</td>
</tr>
<tr>
<td>300</td>
<td>Joshua</td>
<td>3</td>
</tr>
<tr>
<td>400</td>
<td>Nicholas</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5-7: Salesman Table with smin=1, smax=1, rmin=1, rmax=1

5.4.2 One-to-Many Relationships

In a one-to-many relationship, a foreign key reference to the table on the “many” side of the relationship must be present in the table on the “one” side of the relationship. In our example, we will assume that the Region table is on the “many” side of the relationship, and the Salesman table is on the “one” side of the relationship. That being the case, the schemas for the two tables need not change from those described for the one-to-one relationship (see Table 5-2 and Table 5-3).
Once again, the data for the Salesman table could be generated in a number of ways, depending upon the desired participation level of the Salesman and Region tables.

### 5.4.2.1 LHS partially participating, RHS partially participating

- LHS cardinality constraint: 0..1
- RHS cardinality constraint: 0..N

In this situation, using our Salesman/Region example, the Salesman table could be modeled in SDDL as follows:

```xml
<database>
  <seed>7829683417</seed>
  <import filename="names.xml"/>
  <table name="Salesman">
    <field name="ID" type="int">
      <iteration base="1" count="10"/>
    </field>
    <field name="Name" type="CHAR(16)">
      <formula>names['Caucasian'].firstnames</formula>
    </field>
    <variable name="rID" type="int">
      <queryPool query="SELECT ID FROM Region"/>
    </variable>
    <field name="RegionID" type="int">
      <formula>(IRND(2)==0)?"NULL":rID</formula>
    </field>
  </table>
</database>
```

In SDDL table definition above, 10 entries are generated for the Salesman table. The RegionID field may either be NULL or some valid ID value from the Region table. The generated table looks like this:
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>RegionID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joseph</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>Anthony</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>Joshua</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Nicholas</td>
<td>NULL</td>
</tr>
<tr>
<td>5</td>
<td>Olivia</td>
<td>NULL</td>
</tr>
<tr>
<td>6</td>
<td>Ethan</td>
<td>NULL</td>
</tr>
<tr>
<td>7</td>
<td>Abigail</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>Andrew</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Nicholas</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>Lauren</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5-8: Salesman Table with \( s_{min}=0, s_{max}=1, r_{min}=0, r_{max}=(N>1) \)

The table definition allowed for partial participation by each entity, and indeed each entity partially participates in this instance (some RegionID values are NULL, and not all RegionID values are represented). The generated data also reflects the one-to-many relationship, in that the Salesman entries participate at most once in the relationship, and the Region entries can participate more than once.

5.4.2.2 LHS fully participating, RHS partially participating

- LHS cardinality constraint: 1..1
- RHS cardinality constraint: 0..N

In this situation, using our Salesman/Region example, we could describe the Salesman table in SDDL as follows:

```xml
<database>
    <seed>7829683417</seed>
    <import filename="names.xml"/>
    <table name="Salesman">
        <field name="ID" type="int">
            <iteration base="1" count="10"/>
        </field>
        <field name="Name" type="CHAR(16)"/>
    </table>
</database>
```
This fairly simple description ensures that each entity in the Salesman table participates once and only once in the relationship, and that each entity in the Region table participates any number of times in the relationship. This definition produces a Salesman table that looks like this:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>RegionID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joseph</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Anthony</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Joshua</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Nicholas</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Olivia</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Ethan</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>Abigail</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>Andrew</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Nicholas</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>Lauren</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5-9: Salesman Table with smin=1, smax=1, rmin=0, rmax=(N>1)

5.4.2.3 LHS partially participating, RHS fully participating

- LHS cardinality constraint: 0..1
- RHS cardinality constraint: 1..N

In this situation, using our Salesman/Region example, we could define the Salesman table in SDDL as follows:
The rID iteration assures that each record in the Region table will be visited at least once, and possibly 2 or three times. The counter iteration variable is used to insure that each ID in the Region table will participate at least once in the relation. On the second or third repetition of an element in the rID iteration, a NULL value might be produced for RegionID, or the RegionID might be repeated. The NULL values enable partial participation by the Salesman table, and the repeated RegionIDs enable multiple participation by the Region table. The Salesman table generated looks like this:
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>RegionID</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Elizabeth</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Olivia</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Jessica</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Tyler</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>John</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>Zachary</td>
<td>NULL</td>
</tr>
<tr>
<td>14</td>
<td>Emily</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5-10: Salesman Table with smin=0, smax=1, rmin=1, rmax=(N>1)

5.4.2.4 LHS fully participating, RHS fully participating

- LHS cardinality constraint: 1..1
- RHS cardinality constraint: 1..N

In this situation, our example Salesman table could be described in SDDL as follows:

```xml
<database>
  <seed>7829683417</seed>
  <import filename="names.xml"/>
  <table name="Salesman">
    <variable name="rID" type="int">
      <iteration query="SELECT ID FROM Region">
        <repeatMin>1</repeatMin>
        <repeatMax>3</repeatMax>
        <itervar name="counter" type="int" init="0"/>
      </iteration>
    </variable>
    <field name="ID" type="int">
      <formula>rID*3+counter</formula>
    </field>
    <field name="Name" type="CHAR(16)">
      <formula>names['Caucasian'].firstnames</formula>
    </field>
    <field name="RegionID" type="int">
      <formula>rID</formula>
    </field>
    <variable name="incCounter" type="int">
      <formula>counter=counter+1</formula>
    </variable>
  </table>
</database>
```
The definition above does not produce NULL values, so all rows in the Salesman table will participate in the “Sells In” relationship. Also, each record in the Region table is visited at least once, and possibly 2 or 3 times, so each ID in the Region table participates 1 or more times in the “Sells In” relation. The table generated from this definition might look like this:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>RegionID</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Elizabeth</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Olivia</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Andrew</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Tyler</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>John</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>Emily</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>Zachary</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5-11: Salesman Table with $s_{min}=1$, $s_{max}=1$, $r_{min}=1$, $r_{max}=(N>1)$

5.4.3 Many-to-Many Relationships

To manage the data associated with many-to-many relationships, it is necessary to create a third table with foreign keys referencing the primary keys of the two related tables. For our Salesman/Region example, we would need to implement the following schema to handle many-to-many relationships between the Salesman and Region tables:

<table>
<thead>
<tr>
<th>Salesman</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID: INTEGER</td>
</tr>
</tbody>
</table>

Table 5-12: Layout for Salesman Table in Many-to-Many Relationship
Table 5-13: Layout for Region Table in Many-to-Many Relationship

<table>
<thead>
<tr>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID: INTEGER</td>
</tr>
<tr>
<td>Name: CHAR(5)</td>
</tr>
</tbody>
</table>

The additional SalesmanRegion table provides the necessary linkage between the Salesman and Region tables in a many-to-many relationship.

For many-to-many relationships, it will be assumed that the Region and Salesman tables already exist. We will assume a Region table identical to Table 5-1, and a Salesman table that looks like this:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Matthew</td>
</tr>
<tr>
<td>2</td>
<td>James</td>
</tr>
<tr>
<td>3</td>
<td>Ethan</td>
</tr>
<tr>
<td>4</td>
<td>Olivia</td>
</tr>
<tr>
<td>5</td>
<td>Ashley</td>
</tr>
<tr>
<td>6</td>
<td>William</td>
</tr>
<tr>
<td>7</td>
<td>Ethan</td>
</tr>
<tr>
<td>8</td>
<td>David</td>
</tr>
<tr>
<td>9</td>
<td>Joseph</td>
</tr>
<tr>
<td>10</td>
<td>Brandon</td>
</tr>
</tbody>
</table>

The challenge, then, will be to generate the SalesmanRegion table. This table can be generated in a number of ways, depending upon the desired participation constraints of the Salesman and Region tables.
5.4.3.1 LHS partially participating, RHS partially participating

- LHS cardinality constraint: 0..M
- RHS cardinality constraint: 0..N

In this situation, using our Salesman/Region/SalesmanRegion example, we could define the SalesmanRegion table in SDDL as follows:

```
<database>
  <seed>7829683417</seed>
  <table name="SalesmanRegion">
    <field name="sID" type="int">
      <iteration query="SELECT ID FROM Salesman">
        <repeatMin>0</repeatMin>
        <repeatMax>1</repeatMax>
      </iteration>
    </field>
    <field name="rID" type="int">
      <iteration query="SELECT ID FROM Region">
        <repeatMin>0</repeatMin>
        <repeatMax>1</repeatMax>
      </iteration>
    </field>
  </table>
</database>
```

Using such a table definition, each ID in Salesman (sID) is guaranteed to participate 0 or more times, and each ID in Region (rID) is guaranteed to participate 0 or more times. The table generated from the definition might look like this:

<table>
<thead>
<tr>
<th>sID</th>
<th>rID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 5-16: SalesmanRegion Table where $smin=0, \ smax=(M>1), \ rmin=0, \ rmax=(N>1)$

In this case, the Region table fully participates in the “Sells In” relationship. However, that was not guaranteed by the manner in which the SalesmanRegion table was defined in SDDL. Remember that a partially participating table may still result in a fully participated instance of the table.

5.4.3.2 Guaranteeing Full Participation in a Many-to-Many Relationship

When defining the third table in SDDL so as to ensure full participation by one side or the other in a many-to-many relationship, the process tends to take on a stochastic nature. Here is a template for defining full participation by one side or the other in a table $AB$ that relates tables $A$ and $B$ based on the ID fields of those tables (and at the same time avoids the repetition of primary key values in the $AB$ table):

```xml
<database>
  <seed>7829683417</seed>
  <table name="AB">
    <field name="aID" type="int">
      <iteration query="SELECT ID FROM A"/>
    </field>
    <field name="bID" type="int">
      <iteration query="SELECT ID FROM B">
        <repeatDist>
          <tier prob="probMiss" min="0" max="0"/>
          <tier prob="probHit" min="1" max="1"/>
        </repeatDist>
      </iteration>
    </field>
  </table>
</database>
```
The probHit value is the probability that a single element in the bID iteration will generate a value. The probMiss value is the probability that a single element in the bID iteration will not generate a value. Both probHit and probMiss are between 0 and 1, inclusive, and probHit+probMiss=1.0.

In order for a single A element to not participate in the relationship, all elements in the inner B iteration must result in no output. The probability (P_Amiss) of this happening is:

\[ P_{A\text{miss}} = \text{probMiss}^{|B|} \]

In order for A to fully participate in the relationship, all elements of A must participate in the relationship. Since the probability of an individual A element participating in the relationship is (1-P_Amiss), the probability (P_Afull) of all elements of A participating in the relationship is therefore:

\[ P_{A\text{full}} = (1-P_{A\text{miss}})^{|A|} = (1 - \text{probMiss}^{|B|})^{|A|} \]

The probability (P_Bfull) of table B fully participating in the relationship is defined by equations symmetric to the ones above:

\[ P_{B\text{miss}} = \text{probMiss}^{|A|} \]

\[ P_{B\text{full}} = (1-P_{B\text{miss}})^{|B|} = (1 - \text{probMiss}^{|A|})^{|B|} \]

139
What kind of values should we use for probHit and probMiss? That depends upon the cardinalities of the participating tables, as well as the participation goals for the two tables. Suppose that we want to pick probMiss and probHit for our concrete example, the “Sells In” relation. Recall that |Salesman| = 10 and |Region| = 4. The following table shows the effects of various values of probHit and probMiss for this situation (“A” is the Salesman table, “B” is the Region table):

<table>
<thead>
<tr>
<th>probHit</th>
<th>probMiss</th>
<th>P_{A\text{full}}</th>
<th>P_{B\text{full}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.9</td>
<td>2.31378E-05</td>
<td>0.179962417</td>
</tr>
<tr>
<td>0.2</td>
<td>0.8</td>
<td>0.005145925</td>
<td>0.634859723</td>
</tr>
<tr>
<td>0.3</td>
<td>0.7</td>
<td>0.064204349</td>
<td>0.891707916</td>
</tr>
<tr>
<td>0.4</td>
<td>0.6</td>
<td>0.249567957</td>
<td>0.976032016</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>0.524460475</td>
<td>0.996099468</td>
</tr>
<tr>
<td>0.6</td>
<td>0.4</td>
<td>0.771565416</td>
<td>0.999580636</td>
</tr>
<tr>
<td>0.7</td>
<td>0.3</td>
<td>0.921889572</td>
<td>0.999906352</td>
</tr>
<tr>
<td>0.8</td>
<td>0.2</td>
<td>0.98411471</td>
<td>0.999999999</td>
</tr>
<tr>
<td>0.9</td>
<td>0.1</td>
<td>0.99900045</td>
<td>1</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5-17: Effects of various probHit, probMiss values on SalesmanRegion Table

If we pick a probHit of 0.9, then there is a very good chance that both tables will fully participate in the relation. If we pick a probHit of 0.4, then there is a reasonably good chance that the Region table will fully participate in the relation, and that the Salesman table will **not** fully participate in the relation. Let us put this to the test.
The following SDDL table definition follows our template for generating the SalesmanRegion table with probHit of 0.4 and probMiss of 0.6:

```xml
<database>
  <seed>7829683417</seed>
  <table name="SalesmanRegion">
    <field name="sID" type="int">
      <iteration query="SELECT ID FROM Salesman"/>
    </field>
    <field name="rID" type="int">
      <iteration query="SELECT ID FROM Region">
        <repeatDist>
          <tier prob="0.6" min="0" max="0"/>
          <tier prob="0.4" min="1" max="1"/>
        </repeatDist>
      </iteration>
    </field>
  </table>
</database>
```

The table generated by the SDDL definition above looks like this:

<table>
<thead>
<tr>
<th>sID</th>
<th>rID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5-18: SalesmanRegion Table generated with probHit=0.4, probMiss=0.6

Indeed, the SalesmanRegion table generated with probHit=0.4 and probMiss=0.6 results in the Salesman table partially participating in the relationship and the Region table fully participating in the relationship.
In a general sense, the following hold true regarding our many-to-many table generation template:

- If $|A|$ and $|B|$ are roughly equal, then there will be no value of $\text{probHit}$ that assures (or even makes it likely) that one side fully participates in the relationship and the other side partially participates.

- If $|A| >> |B|$, there is a $\text{probHit}$ value that makes it probable that $B$ will fully participate and $A$ will partially participate. However, there is no value of $\text{probHit}$ that will make it likely that $A$ fully participates and $B$ partially participates.

- If $|B| >> |A|$, there is a $\text{probHit}$ value that makes it probable that $A$ will fully participate and $B$ will partially participate. However, there is no value of $\text{probHit}$ that will make it likely that $B$ fully participates and $A$ partially participates.

- The larger the values of $|A|$ and $|B|$, the smaller the $\text{probHit}$ value that must be chosen to make it likely that $A$ or $B$ partially participates in the relationship.

- The use of $\text{probHit}$ and $\text{probMiss}$ in this fashion only makes various outcomes statistically likely or unlikely. There is no guarantee of partial or full participation for any $\text{probHit}/\text{probMiss}$ values other than 0 or 1.

Of course, partial participation from each side is easy to model, as shown in section 5.4.3.1.
5.4.4 N-ary Relationships

All of our previous examples of enforcing cardinality constraints involved binary relationships. General n-ary relationships can also be modeled and enforced by SDDL, using similar techniques as those employed for the binary relationships. In a general sense, an n-ary relationship implies n-1 or n foreign keys in the derived table. To generate foreign key values for such a table, use QueryIterations or QueryPools in the same manner as that demonstrated for binary relationships.

5.5 Weak Entity Support

In the E-R Model, entity A is said to have an exclusive or identifying relationship ([22], p. 77) with entity B if an entity in A cannot exist in a relationship without an entity in B. Entities with exclusive relationships are sometimes referred to as weak entities.

![Figure 5-2: Weak Entity Example](image1)

![Figure 5-3: Weak Entity Specified Through Cardinality Constraints](image2)
As an example, consider the E-R diagram from Figure 5-2. The Parent relationship relates employees and dependents. A dependent would not exist in this relationship without an employee, so the Dependent entity is a weak entity (and is thus represented by the double-box). There is such a thing, however, as an employee without dependents, and so the Employee entity is not an exclusive or weak entity.

The idea of a weak entity really just reduces to cardinality constraints. If an entity is exclusive (or weak), then the minimum cardinality of the entity is 1, meaning that each entity item participates at least once in the relationship. The minimum cardinality of the related non-exclusive entity is 0, meaning that not all entity items are required to participate in the relationship. Thus, the weak entity fully participates in the relationship, and the related non-exclusive entity partially participates in the relationship. Figure 5-3 shows the weak entity example from Figure 5-2 described using cardinality constraints.

The maximum cardinalities of the entities participating in an exclusive relationship will vary among relationships. The relationship may be one-to-one, one-to-many, or many-to-many. Regardless, techniques for describing all combinations of cardinality constraints have been given in Section 5.4. Using those techniques, SDDL can be used to enforce the concept of weak entities.

5.6 A Word on Circular Data Dependencies

SDDL exhibits a linear nature when describing inter-table data dependencies. If table B has data that depends upon the values in table A, then table A must be generated first and referenced by table B. This linear dependency resolution logic means that SDDL cannot easily be used to resolve circular inter-table data dependencies.
For an example, consider the TPC-H benchmark [23]. Some of the attributes and dependencies associated with the TPC-H LINEITEM and ORDERS tables are shown in Figure 5-4. The LINEITEM table cannot be generated without the ORDERS table, because the LINEITEM.L_ORDERKEY attribute is a foreign key reference to the ORDERS.O_ORDERKEY attribute, and the L_SHIPDATE, L_COMMITDATE and L_RECEIPTDATE attributes of the LINEITEM table depend upon the value of ORDERS.O_ORDERDATE. However, the ORDERS table cannot be generated without the LINEITEM table, because the ORDERS.O_TOTALPRICE attribute cannot be calculated without knowing the contents of the LINEITEM table. Thus, it is not possible for SDDL to explicitly model this type of relationship.

Figure 5-4: Circular Dependencies in TPC-H Tables

The most straightforward way to resolve such a problem is to generate a temporary L_ORDERDATE attribute for each row in the LINEITEM table; entries with identical L_ORDERKEY values will also have identical L_ORDERDATE values. The L_COMMITDATE, L_SHIPDATE and L_RECEIPTDATE fields will then depend only upon L_ORDERDATE; in this fashion, the inter-table dependencies involving
L_COMMITDATE, L_SHIPDATE and L_RECEIPTDATE can be eliminated. The ORDERS table can then be generated after the LINEITEM table, and the ORDERS.O_ORDERDATE value for each O_ORDERKEY entry can be set to the common L_ORDERDATE value for that order key. After the generation is complete, the extraneous LINEITEM.L_ORDERDATE field can be eliminated.

In this fashion, “scaffolding” can be constructed during generation, much like scaffolding is present during the construction of a building. While the data is being generated, extra columns are generated in order to resolve circular inter-table dependencies. After the need for such scaffolding has passed, the extra columns can be removed, leaving the generated data in its pristine format.

The following SDDL could be used to generate the LINEITEM table with such scaffolding:

```xml
<table name="LINEITEM">
  <field name="L_ORDERKEY" type="int">
    <iteration base="10000" count="100">
      <itervar name="orderdate" type="date" init="#1992-01-01#+IRND(7*365-151)"/>
    </iteration>
  </field>
  <field name="L_LINENUMBER" type="int">
    <iteration base="1" count="1+IRND(7)"/>
  </field>
  <field name="L_SHIPDATE" type="date">
    <formula>orderdate+(1+IRND(121))</formula>
  </field>
  <field name="L_COMMITDATE" type="date">
    <formula>orderdate+(30+IRND(61))</formula>
  </field>
  <field name="L_RECEIPTDATE" type="date">
    <formula>L_SHIPDATE+(1+IRND(30))</formula>
  </field>
  <!-- Extra Column -->
  <field name="L_ORDERDATE" type="date">
    <formula>orderdate</formula>
  </field>
</table>
```
The ORDERS table could then be described based on the LINEITEM table as follows:

```xml
<field name="O_ORDERKEY" type="int">
  <iteration query="SELECT L_ORDERKEY, min(L_ORDERDATE) odate, sum(L_QUANTITY* L_EXTENDEDPRICE* (1-L_DISCOUNT)* (1+L_TAX)) total from LINEITEM group by L_ORDERKEY"/>
</field>
<field name="O_ORDERDATE" type="date">
  <formula>DATE(O_ORDERKEY$odate)</formula>
</field>
<field name="O_TOTALPRICE" type="NUMERIC(9,2)">
  <formula>0.0+O_ORDERKEY$total</formula>
</field>
```

The ORDERS table has now effectively taken its O_ORDERKEY and O_ORDERDATE values from the LINEITEM table, effectively breaking the circular dependency. After the ORDERS table has been generated, the LINEITEM.L_ORDERDATE column may be deleted.

5.7 Summary

In this chapter, it has been shown that SDDL is capable of modeling the common constraint types found in the relational and E-R models. SDDL is capable of enforcing key constraints, entity integrity constraints, domain constraints and referential integrity
constraints. It can also effectively model all combinations of cardinality constraints, for both unary and binary relationships. While SDG/SDDL cannot explicitly model circular inter-table data dependencies, such circular dependencies can often be eliminated through the generation of temporary attributes ("scaffolding") for a table.
6 RANDOM BUT DETERMINISTIC

In Chapter 4, it was shown that SDG produced deterministic data sets (given a constant input) regardless of the degree of parallelism employed during generation. This property of determinism makes SDG useful for regression testing applications. However, within the confines of user-defined constraints and the necessity for determinism, it is desirable that the generated data be as random as possible in order to appear realistic, to avoid injecting unwanted patterns not present in corresponding real data, and to avoid re-generating already generated data. This chapter investigates the question, “Does SDG produce sufficiently random data?”

6.1 Background

6.1.1 Definitions of Terms

*RNG*: Random Number Generator

*LCG*: Linear Congruential Generator. An LCG is an RNG defined by a recurrence relation of the form

\[ X_{i+1} = (A \times X_i + B) \mod M \]

The value for term \(X_{i+1}\) depends upon the value for term \(X_i\). A specific LCG is defined by its definitions of the constants \(A\), \(B\) and \(M\).

*Seed*: A random number seed is a number or vector of numbers used to initialize a random number generator [24]. Given a seed as a starting point, the random number generator will produce a deterministic sequence of random numbers. In the LCG equation above, the \(X_i\) term serves as a seed for the \(X_{i+1}\) term.
**MRG:** Multiple Recursive Generator. An MRG is a generalization of an LCG. An LCG has a single recurrence term in its recurrence relation, while an MRG can have multiple recurrence (X) terms in its recurrence relation. An MRG might have a recurrence equation that looks something like this:

\[
X_{i+1} = (A \times X_i + B \times X_{i-1} + C \times X_{i-2} + D) \mod M
\]

**CMRG:** Combined Multiple Recursive Generator. A CMRG combines the results of two or more MRGs to generate random numbers.

**Period:** The period of a random number generator is the cardinality of the set of random (or pseudo-random) numbers through which the generator will cycle before repeating itself.

### 6.1.2 A Discussion of Streams

Recall from Chapter 4 the method by which SDG can generate consistent data sets (given a consistent input) regardless of the degree of parallelism employed during data generation. For data descriptions that do not contain iteration constraints, the output is divided into blocks of 100 rows each and these blocks are evenly distributed among the generation processes. Each block of 100 rows can be considered a *stream*. For data descriptions that do contain iterations, the output is divided into the output associated with each outer iteration element and these chunks are evenly distributed among the generation processes. Each of these chunks can be also considered a *stream*. Therefore, regardless of the presence or absence of iteration constraints, the output data can be considered in an abstract way to be divided into a number of streams. In general, a generation process generates data as follows:
Process k of N:
  RF(seed, k)
  Generate stream k
  RF(seed, k+N)
  Generate stream k+N
  RF(seed, k+2N)
  Generate stream k+2N ...

The reseed function $RF()$ serves to reset the underlying random number generator to a deterministic starting point given the user-specified seed and the sequence number of the upcoming stream. Regardless of the number of participating generation processes, the careful use of the reseed function $RF()$ will guarantee that each individual stream is initialized deterministically and generated identically from run to run. The reseed function $RF()$ is therefore used to implement a multi-stream random number generator.

However, if one is not careful, the reseed logic could cause portions of previous streams to be regenerated. Suppose that one needed to generate four streams of data. Figure 6-1, below, is an example of badly chosen starting points for four streams (A-D). Stream C repeats some of the values from stream A, and stream D repeats some of the values from stream B. Such repetition between two streams may result in unintended correlation between the values output by the two streams, engendering non-random behavior in the data set produced.
Figure 6-1: Badly chosen stream starting points

Optimally, if one wants to produce N streams, the starting points for the streams should be evenly distributed across the period of the generator, each starting one-\(N^{th}\) of the period apart from its neighbors. In Figure 6-2, the four sample streams are each started one-fourth of the period apart from its neighbors. While this does not preclude repetition among streams (consider what would happen if the N streams were longer than one-\(N^{th}\) of the period), it statistically minimizes the probability of that occurring.
6.1.3 Streaming Mechanisms

The choice of random streaming mechanism (i.e., the abstract reseed function $RF$ mentioned earlier) is critical in implementing a synthetic data generator. In this subsection, two streaming mechanisms are described, and their effectiveness is measured and compared later in this chapter.

6.1.3.1 The Linear Streaming Mechanism (LSM)

The Linear Streaming Mechanism (LSM) is a simple, fast streaming mechanism based on the Java random number generator. To begin producing the random numbers associated with stream $streamIndex$, LSM simply uses $13927^8$ as a multiplier and 

---

There is no significance to the number 13927. It’s not even prime! The author pulled it out of the air when first coding the generator; it appealed to him as a reasonably large odd number. The lack of science involved in the development of the original streaming mechanism gave rise to concerns about the randomness of the mechanism and resulted in the writing of this chapter.
the user-defined seed as an offset, as follows (where `Globals.rnd` is an object of type `java.util.Random`):

```java
ReseedFunction(userSeed, streamIndex)
{
   Globals.rnd.setSeed(userSeed+streamIndex*13927);
    Globals.rnd.nextInt();
}
```

According to the Java documentation [25], the Java RNG is an LCG that uses this logic:

```java
synchronized protected int next(int bits) {
    seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    return (int)(seed >>> (48 - bits));
}
```

Thus, the LCG formula for the Java RNG becomes:

\[ X_{i+1} = (25214903917 \times X_i + 11) \mod 2^{48} \]

In practice, LCGs do an acceptable job of producing random number sets [26]. However, they do exhibit some weaknesses [26]:

- The low-order bits tend to be non-random, especially if \( M \) is set to a power of 2 (as is the case in the Java RNG). This is the reason that the `next()` method above returns the high-order bits of the 48-bit value produced by the recurrence relation. The higher order bits tend to be more random, and the lower order bits tend to be less random.

---

9 Note that a leading “0x” denotes a hexadecimal number, a trailing “L” denotes a long integer, and the “<<” operator stands for “shift left”.

154
• Streams produced by LCGs (especially if $A$, $B$ and $M$ are chosen badly) are subject to hyperplaning. A hyperplane [27] in N-dimensional space is an (N-1)-dimensional object that divides that space into two spaces. A badly configured LCG might produce 2-dimensional data points as shown in Figure 6-3 below, which is clearly not random.

![Hyperplaning in 2-dimensional space](image)

Figure 6-3: Hyperplaning in 2-dimensional space

In addition to the potential for hyperplaning, it was unclear whether the streaming imposed by `ReseedFunction()` (an implementation of the abstract reseed function `RF()`) would result in evenly distributed streams as in Figure 6-2 or bunched streams as in Figure 6-1. Thus, the simplicity of LSM was attractive, but its ability to generate evenly distributed random streams was suspect.

6.1.3.2 CMRG Streaming Mechanism (CSM)

The CMRG Streaming Mechanism is based on Pierre L’Ecuyer’s “RngStream” random number generation Java class [28]. RngStream is not an LCG; instead, it is a combined multiple recursive generator (CMRG). L’Ecuyer’s RngStream class is based on the MRG32k3a CMRG [29], which uses the following MRGs:
\[
X_n = (1403580 \times X_{n-2} - 810728 \times X_{n-3}) \mod m_1 \\
Y_n = (527612 \times Y_{n-1} - 1370589 \times Y_{n-3}) \mod m_2
\]

where \( m_1 = 2^{32} - 209 = 4294967087 \) and \( m_2 = 2^{32} - 22853 = 4294944443 \). The output \((U_n)\) is a real number between 0 and 1 defined by

\[
Z_n = (X_n - Y_n) \mod 4294967087 \\
U_n = Z_n / 4294967088
\]

From [30]: “Combining the two multiple recursive generators (MRG) results in sequences with better statistical properties in high dimensions and longer periods compared with those generated from a single MRG. The combined generator … has a period length of approximately \(2^{191} \).”

RngStream is specifically designed to support independent streams of random data. Calling the resetNextSubstream() method of RngStream will cause an entirely new random number stream to be started. According to [31], “The successive streams actually start \( Z = 2^{127} \) steps apart, and each stream is partitioned into \(2^{51}\) adjacent substreams of length \( W = 2^{76} \).” In other words, given a random seed, there are \(2^{51}\) independent substreams available to the user, each of which is \(2^{76}\) long. For most practical purposes, then, RngStream provides an infinite number of independent substreams each of which is infinitely long.

6.2 A Test for Randomness

The dictionary (dictionary.reference.com) defines random as “proceeding, made, or occurring without definite aim, reason, or pattern: the random selection of numbers.” For the purposes of this research, the word “pattern” is important; we want to be sure that
our random number streams do not exhibit predictable patterns. How would one go about testing the randomness of a stream of numbers?

Sometimes, one can detect non-randomness by simply looking at the data. For example, these streams are easily detectable as non-random:

- 0,0,0,0,0,0,0,0 (all zeroes)
- 1,2,3,4,5,6,7,8 (up-counting)
- 0,1,0,1,0,1,0,1 (alternating zeroes and ones)

However, it is more often the case that a non-random stream of numbers will look random to the casual observer – especially if the stream is very long. In other words, a stream of numbers may exhibit a pattern that is not easily discernable, and thus be non-trivially non-random. A rigorous mathematical test is therefore needed to determine randomness in many cases.

The National Institute of Standards and Technology (NIST) has developed a Statistical Test Suite (STS) [32, 33] for just this purpose – testing the randomness of a stream of numbers. STS applies up to 15 different tests to a stream of binary bits. These 15 tests can be briefly summarized as follows (taken from [32]):

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (Monobits) Test</td>
<td>Test that the number of zeroes and ones in the binary bit stream are approximately equal.</td>
</tr>
<tr>
<td>Test for Frequency within a Block</td>
<td>Performs 0/1 frequency test within each M-bit block of data.</td>
</tr>
<tr>
<td>Runs Test</td>
<td>Analyzes the number and lengths of 0-runs (uninterrupted streams of 0s) and 1-runs within the stream.</td>
</tr>
<tr>
<td>Test for Longest Run of Ones in a Block</td>
<td>Tests for the longest run of ones within each M-bit block.</td>
</tr>
<tr>
<td>Random Binary Matrix Test</td>
<td>Check for linear dependence among fixed length substrings of the binary stream.</td>
</tr>
<tr>
<td>Discrete Fourier Transform (Spectral) Test</td>
<td>Detect periodic features in the binary stream.</td>
</tr>
<tr>
<td>Test Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Non-Overlapping Template Matching Test</td>
<td>Checks for the number of occurrences of a number of m-bit patterns in the binary stream. When a match is found, the search window is reset to the bit after the matched pattern.</td>
</tr>
<tr>
<td>Overlapping Template Matching Test</td>
<td>Checks for the number of occurrences of a number of m-bit patterns in the binary stream. When a match is found, the search window slides one bit.</td>
</tr>
<tr>
<td>Maurer’s Universal Statistical Test</td>
<td>Detect whether or not the binary stream can be significantly compressed without loss of information. An overly compressible sequence is considered to be non-random.</td>
</tr>
<tr>
<td>Linear Complexity Test</td>
<td>Determine whether or not the binary stream is complex enough to be considered random.</td>
</tr>
<tr>
<td>Serial Test</td>
<td>Tests the frequency of each and every overlapping m-bit pattern across the bit stream.</td>
</tr>
<tr>
<td>Approximate Entropy Test</td>
<td>Compares the frequency of overlapping blocks of two consecutive/adjacent lengths (m and m+1) against the expected result for a random sequence.</td>
</tr>
<tr>
<td>Cumulative Sum Test</td>
<td>Interprets 1s as 1s, and 0s as -1s, and cumulatively adds up the entire binary stream. Random streams should never stray too far from a 0 sub-sum.</td>
</tr>
<tr>
<td>Random Excursions Test</td>
<td>Tests the number of cycles having exactly K visits in a cumulative sum random walk test.</td>
</tr>
<tr>
<td>Random Excursions Variant Test</td>
<td>Tests the number of times that a particular state occurs in a cumulative sum random walk test.</td>
</tr>
</tbody>
</table>

Table 6-1: Tests applied by NIST STS software (adapted from [32])

STS produces a $p$-value for each test. If the $p$-value for a stream is more than 0.01 for a given test, then the stream is considered random with respect to that test. Otherwise, the stream is considered non-random with respect to that test. It is often the case that a stream of bits will pass some tests and fail others.

### 6.3 Randomness Test Results

Both the Linear Streaming Mechanism (LSM) and CMRG Streaming Mechanism (CSM) were run through the NIST test suite. Since it is hard to predict the streaming pattern of arbitrary synthetic data, two different tests were run for each: (1) Lots of small streams, generating 10,000 streams of 48 24-bit numbers, and (2) A few large streams,
generating 100 streams of 4,800 24-bit numbers. In each case, a total of 11,520,000 bits were generated and interpreted as a single bit stream.

All 15 STS sub-tests were applied during each test. Default settings were used for each test, with the exception of the Universal Test, where the size of the stream being analyzed dictated the use of values 10 and 10240 for “block length” and “number of initialization steps”, respectively.

6.3.1 Producing the Random Number Streams

In order to run our tests, it was necessary to produce random number streams from both LSM and CSM. Java code was written for this purpose. This subsection provides details regarding that Java code.

6.3.1.1 RngInterface

Each streaming mechanism was written to conform to the following interface:

```java
public interface RngInterface {
    public void setRefSeed(long seed);
    public void changeToStream(int streamNo);
    public int nextInt(int uLim);
    public double nextDouble();
}
```

The `setRefSeed()` method allows the user to choose a reference seed for the generator. The `changeToStream()` method shifts output to the indicated random stream number. The `nextInt()` method returns an random integer between 0 and `uLim` (inclusive), and the `nextDouble()` method returns a double between 0.0 and 1.0.
6.3.1.2 LSMRng

The LSMRng class is defined as follows:

```java
class LSMRng implements RngInterface {
    private java.util.Random rnd;
    private long refSeed = 0;

    public LSMRng() {
        rnd = new java.util.Random();
    }

    public void setRefSeed(long seed) {refSeed = seed;}

    public void changeToStream(int streamNo) {
        rnd.setSeed(refSeed + streamNo * 13927);
        rnd.nextInt();
    }

    public int nextInt(int uLim) {return rnd.nextInt(uLim);}

    public double nextDouble() {return rnd.nextDouble();}
}
```

Note that the LSMRng class uses the java RNG, and the `changeToStream()` method implements our simple stream algorithm.

6.3.1.3 CSMRng

The CSM was implemented as follows:

```java
class CSMRng implements RngInterface {
    private RngStream RNGS;
    private int prevStream = 0;
    private long refSeed;
```
public CSMRng() { RNGS = new RngStream(); }

public void setRefSeed(long seed)
{
    refSeed = seed;
    long[] seedArray =
        new long[]
            {seed,seed,seed,seed,seed,seed};
    RNGS.setSeed(seedArray);
}

public void changeToStream(int streamNo)
{
    int skips = streamNo-prevStream;
    while(skips>0)
    {
        RNGS.resetNextSubstream();
        skips--;
    }
    //RNGS.advanceState(0, streamNo-prevStream);
    prevStream = streamNo;
}

public int nextInt(int uLim) {return
    RNGS.randInt(0,uLim-1);}

    public double nextDouble() { return
    RNGS.randU01();}
}

Note that the CSMRng class is based on the RngStream class/generator, and that
the changeToStream() method will call RNGS.resetNextSubstream() once
for each stream skipped since the previously specified stream number. The
changeToStream() method assumes that the streams are accessed in a monotonically
increasing fashion.

6.3.1.4 The Test Code

The method used to generate a binary stream from a given generator follows:
public static void genbits(FileWriter fout, RngInterface rng, int nstreams, int nnums) {
    int streamIdx, numIdx;
    for(streamIdx=0; streamIdx<nstreams; streamIdx++) {
        rng.changeToStream(streamIdx);
        for(numIdx=0; numIdx<nnums; numIdx++) {
            int newNum = rng.nextInt(NUM_MAX+1);
            String newStr = toBase2(newNum,NUM_BITS);
            try {
                fout.write(newStr);
            } catch(Exception e) {
                System.out.println("Write exception: " + e);
                System.exit(-1);
            }
        }
    }
}

NUM_MAX is defined as 0xFFFFFFFF, and NUM_BITS is defined as 24. Note that the
toBase2() method converts the random number into a binary string of 0s and 1s.

For each generator tested, genbits() was called once with
nstreams=10000 and nnums=48 and once with nstream=100 and
nnums=4800.

6.3.2 Performance of Linear Streaming Mechanism

The test results for LSM are shown in Table 6-2.

For the most part, LSM seems to do a good job of producing random streams.
Nearly every random number generator will fail a small number of the “Non-Overlapping
Template Matching Tests”, so failing a few of those templates is no great cause for alarm.
However, the fact that the large-number-of-small-streams instance failed the “Discrete Fourier Transform (Spectral) Test” is disturbing. This test is specifically designed to detect the “hyperplaning” described in section 6.1.3. It appears, then, that either the Java RNG or the LSM built on top of it (or a combination of both) truly is subject to hyperplaning effects.

<table>
<thead>
<tr>
<th>Test</th>
<th>10000 streams of 48x24 bits</th>
<th>100 streams of 4800x24 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (Monobits) Test</td>
<td>p-value=0.732969</td>
<td>p-value=0.525287</td>
</tr>
<tr>
<td>Test for Frequency within a Block</td>
<td>p-value=0.832437</td>
<td>p-value=0.241244</td>
</tr>
<tr>
<td>Runs Test</td>
<td>p-value=0.345194</td>
<td>p-value=0.209480</td>
</tr>
<tr>
<td>Test for Longest Run of Ones in a Block</td>
<td>p-value=0.231356</td>
<td>p-value=0.382115</td>
</tr>
<tr>
<td>Random Binary Matrix Test</td>
<td>p-value=0.308936</td>
<td>p-value=0.601020</td>
</tr>
<tr>
<td>Discrete Fourier Transform (Spectral) Test</td>
<td>p-value=0.000000</td>
<td>p-value=0.727257</td>
</tr>
<tr>
<td>Non-Overlapping Template Matching Test</td>
<td>Failed 4 of 148 templates</td>
<td>Failed 2 of 148 templates</td>
</tr>
<tr>
<td>Overlapping Template Matching Test</td>
<td>p-value=0.466209</td>
<td>p-value=0.125610</td>
</tr>
<tr>
<td>Maurer’s Universal Statistical Test</td>
<td>p-value=0.859772</td>
<td>p-value=0.328048</td>
</tr>
<tr>
<td>Linear Complexity Test</td>
<td>p-value=0.172120</td>
<td>p-value=0.178385</td>
</tr>
<tr>
<td>Serial Test</td>
<td>p-values=0.470235, 0.727910</td>
<td>p-values=0.106144, 0.083163</td>
</tr>
<tr>
<td>Approximate Entropy Test</td>
<td>p-value=0.037869</td>
<td>p-value=0.991538</td>
</tr>
<tr>
<td>Cumulative Sum Test</td>
<td>p-values=0.562402, 0.613696</td>
<td>p-values=0.689104, 0.760658</td>
</tr>
<tr>
<td>Random Excursions Test</td>
<td>Passed 8 of 8 templates</td>
<td>Passed 8 of 8 templates</td>
</tr>
<tr>
<td>Random Excursions Variant Test</td>
<td>Passed 18 of 18 templates</td>
<td>Passed 18 of 18 templates</td>
</tr>
</tbody>
</table>

Table 6-2: STS results on Linear Streaming Mechanism (LSM)
### 6.3.3 Performance of CMRG Streaming Mechanism

The results of the streams generated CSM are shown in Table 6-3. Note that CSM does not fail the Spectral Test, while LSM did (for the 10,000-stream instance).

<table>
<thead>
<tr>
<th>Test</th>
<th>10000 streams of 48x24 bits</th>
<th>100 streams of 4800x24 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (Monobits) Test</td>
<td>p-value=0.775943</td>
<td>p-value=0.057775</td>
</tr>
<tr>
<td>Test for Frequency within a Block</td>
<td>p-value=0.218003</td>
<td>p-value=0.231010</td>
</tr>
<tr>
<td>Runs Test</td>
<td>p-value=0.330927</td>
<td>p-value=0.474680</td>
</tr>
<tr>
<td>Test for Longest Run of Ones in a Block</td>
<td>p-value=0.618522</td>
<td>p-value=0.993619</td>
</tr>
<tr>
<td>Random Binary Matrix Test</td>
<td>p-value=0.619725</td>
<td>p-value=0.715832</td>
</tr>
<tr>
<td>Discrete Fourier Transform (Spectral) Test</td>
<td>p-value=0.238474</td>
<td>p-value=0.875391</td>
</tr>
<tr>
<td>Non-Overlapping Template Matching Test</td>
<td>Failed 1 of 148 templates</td>
<td>Passed 148 of 148 templates</td>
</tr>
<tr>
<td>Overlapping Template Matching Test</td>
<td>p-value=0.153960</td>
<td>p-value=0.417830</td>
</tr>
<tr>
<td>Maurer’s Universal Statistical Test</td>
<td>p-value=0.862480</td>
<td>p-value=0.757975</td>
</tr>
<tr>
<td>Linear Complexity Test</td>
<td>p-value=0.262492</td>
<td>p-value=0.141641</td>
</tr>
<tr>
<td>Serial Test</td>
<td>p-values=0.764560, 0.648244</td>
<td>p-values=0.295575, 0.214842</td>
</tr>
<tr>
<td>Approximate Entropy Test</td>
<td>p-value=0.220806</td>
<td>p-value=0.392464</td>
</tr>
<tr>
<td>Cumulative Sum Test</td>
<td>p-values=0.958284, 0.762304</td>
<td>p-values=0.106025, 0.031797</td>
</tr>
<tr>
<td>Random Excursions Test</td>
<td>Passed 8 of 8 templates</td>
<td>Passed 8 of 8 templates</td>
</tr>
<tr>
<td>Random Excursions Variant Test</td>
<td>Passed 18 of 18 templates</td>
<td>Passed 18 of 18 templates</td>
</tr>
</tbody>
</table>

Table 6-3: STS results for CMRG Streaming Mechanism (CSM)

### 6.4 Summary

Both streaming mechanisms showed acceptable randomness, in the sense that they both passed the majority of NIST’s tests for randomness. The one major difference between the two mechanisms is that the Linear Streaming Mechanism (LSM) appears to
suffer from hyperplaning effects when generating a large number of streams, as evidenced by its failure on the Spectral Test. The CMRG Streaming Mechanism (CSM) suffers no such effects, and incurs only a marginal time penalty with respect to LSM.

In order to preserve backward compatibility, SDG retains LSM as its default streaming mechanism. However, CSM is made available (through a command-line switch) to those users who desire a more rigorous streaming mechanism.
7 GENERATING OBFUSCATED DATA

One important application of SDG (or any synthetic data generation framework) is to produce a “sanitized” version of an existing data set, e.g., a relational table that contains sensitive data. The sanitized table has some of the same characteristics as the original table, but has been stripped of sensitive or personally identifying data that was contained in the original table. We consider the sanitized table to be an obfuscated version of the original table. This ability to generate obfuscated data (based on real data) is useful for generating realistic regression test data sets, and for exporting realistic but not real data to third parties for analysis.

This chapter explores the concept of obfuscation. A spectrum of data obfuscation levels is presented first, along with SDDL techniques for effecting each level of obfuscation. Next, additional methods of obfuscation are presented; these methods include schema obfuscation and cardinality obfuscation. The chapter closes with a warning about data inference.

7.1 A Spectrum for Data Obfuscation

In this section, a spectrum of data obfuscation levels is introduced. The obfuscation levels (in order from “most data conveyed” to “least data conveyed”) are as follows\textsuperscript{10}:

- Level 0: No Obfuscation
- Level 1: Partial Knowledge Reversible Obfuscation

\textsuperscript{10} Levels 1, 2 and 3 were originally identified by Bakken [10].
• Level 2: Process Reversible Obfuscation

• Level 3: Combination Reversible Obfuscation

• Level 4: Statistically Accurate or Dependency Preserving Irreversible Obfuscation

• Level 5: Domain Preserving Irreversible Obfuscation

• Level 6: Opaque Irreversible Obfuscation

Each level removes some additional information from the table.

Each level will be explained in detail in this section. Along with each level of obfuscation, techniques for implementing such obfuscation in SDDL are shown. For all examples in this section, assume the existence of the following reference table (named Patients):

<table>
<thead>
<tr>
<th>ID</th>
<th>DOB</th>
<th>BldType</th>
<th>HIVPOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>'1974-01-30'</td>
<td>'ANEG'</td>
<td>false</td>
</tr>
<tr>
<td>102</td>
<td>'1985-12-29'</td>
<td>'APOS'</td>
<td>true</td>
</tr>
<tr>
<td>103</td>
<td>'1979-03-29'</td>
<td>'ABNEG'</td>
<td>false</td>
</tr>
<tr>
<td>104</td>
<td>'1976-05-20'</td>
<td>'APOS'</td>
<td>true</td>
</tr>
<tr>
<td>105</td>
<td>'1987-08-25'</td>
<td>'ABPOS'</td>
<td>false</td>
</tr>
<tr>
<td>106</td>
<td>'1987-12-22'</td>
<td>'OPOS'</td>
<td>true</td>
</tr>
<tr>
<td>107</td>
<td>'1968-06-20'</td>
<td>'BNEG'</td>
<td>false</td>
</tr>
<tr>
<td>108</td>
<td>'1966-12-25'</td>
<td>'BNEG'</td>
<td>true</td>
</tr>
<tr>
<td>109</td>
<td>'1985-11-17'</td>
<td>'BNEG'</td>
<td>false</td>
</tr>
<tr>
<td>110</td>
<td>'1973-10-02'</td>
<td>'ANEG'</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 7-1: Reference Patients Table

The Patients table is not meant to be complete or realistic. However, it does showcase three different types of attributes: continuously-valued attributes (DOB field), discrete-valued attribute fields (BldType field), and binary attributes (HIVPOS field). This will allow for the demonstration of obfuscation techniques for all three types of attributes.
It should be noted that an obfuscation level can apply to a single field, a set of fields, or a whole table. Typically, obfuscated data will employ different levels of obfuscation for different fields, depending on the sensitivity and significance of each field. However, in the examples given herein, all fields will be obfuscated at the same level (except for the ID field, which always retains its original value).

### 7.1.1 No Obfuscation (Level 0)

It seems self-evident that the weakest form of obfuscation is to simply produce the original data. This can be done using a combination of a query iteration and the “subfield” operator in SDDL, as follows:

```xml
<database>
  <table name="Sanitized0">
    <variable name="ORIG" type="int">
      <iteration query="SELECT ID,DOB,BldType,HIVPOS,Gen,Hgt,Wgt FROM Patients"/>
    </variable>
    <field name="ID" type="int">
      <formula>ORIG$ID</formula>
    </field>
    <field name="DOB" type="date">
      <formula>DATE(ORIG$DOB)</formula>
    </field>
    <field name="BldType" type="CHAR(5)">
      <formula>ORIG$BldType</formula>
    </field>
    <field name="HIVPOS" type="bool">
      <formula>BOOL(ORIG$HIVPOS)</formula>
    </field>
  </table>
</database>
```

The SDDL table description above will simply reproduce the Patients table as the new Sanitized0 table (level 0). Such “non-obfuscation” of an entire collection of
tables is not very useful; it might make sense, though, to pass through original values for individual fields or a subset of the fields via such Level 0 obfuscation.

### 7.1.2 Reversible Obfuscation (Levels 1-3)

Reversible obfuscation techniques have the property that the new value for an attribute is based on the original value for the attribute.

Some applications could benefit from obfuscating data but also from the ability to be able to reverse the obfuscation. For instance, if party A reversibly obfuscated only the ID field in the Patient table and gave the obfuscated data to party B to run a test, then B could run the test and return the reversibly obfuscated IDs to A who could then identify which original IDs passed or failed the test. In the examples that follow, we often focus on leaving the ID field unobfuscated and instead obfuscate the rest of the field so the reader can easily see the changes.

Reversible obfuscation has risks. If one can somehow gain knowledge of the technique used to encode the new value, then one might be able to deduce the original value from the new value. The concept and names of the various levels of reversible obfuscation were taken from Bakken [17].

#### 7.1.2.1 Partial Knowledge Reversible Obfuscation (Level 1)

“An obfuscation technique is partial knowledge reversible if the attacker can reverse-engineer the entire data set using a minimum number of original data set entries.”

[17] Level 1 obfuscation techniques are usually something very simple, such as one of the following:

\[
Value_{obfuscated} = Value_{original} + k
\]
\[ \text{Value}_{\text{obfuscated}} = (\text{Value}_{\text{original}} + k) \times j \]

An attacker who knows a few of the original values can quickly deduce the offset (and/or multiplicand) used to obfuscate the data by comparing known original values to obfuscated values. The attacker can then reverse engineer the remainder of the original value set.

Level 1 obfuscation would typically be used when the threat of attack is judged to be low, or if it is judged unlikely that an attacker has enough partial knowledge to reverse the obfuscation (and the risk of being wrong is modest). We could implement Level 1 obfuscation on all fields in the “Patients” table through the following SDDL table definition:

```xml
<database>
  <pool name="BTs">
    <choice name="APOS"/>
    <choice name="ANEG"/>
    <choice name="BPOS"/>
    <choice name="BNEG"/>
    <choice name="OPOS"/>
    <choice name="ONEG"/>
    <choice name="ABPOS"/>
    <choice name="ABNEG"/>
  </pool>

  <table name="Sanitized1">
    <variable name="ORIG" type="int">
      <iteration
        query="SELECT ID,DOB,BldType,HIVPOS FROM Patients"/>
    </variable>
    <field name="ID" type="int">
      <formula>0+ORIG$ID</formula>
    </field>
    <field name="DOB" type="date">
      <formula>DATE(ORIG$DOB)+73</formula>
    </field>
    <field name="BldType" type="CHAR(5)">
      <formula>BTs[ORIG$BldType]+5</formula>
    </field>
  </table>
</database>
```
In the description above, the obfuscated DOB values were set to 73 days after the original DOB values. The obfuscated BldType values were the original BldType values shifted by 5 in the BTs pool. The obfuscated HIVPOS values were set to the complements of the original HIVPOS values. The results are shown in Table 7-2.

<table>
<thead>
<tr>
<th>ID</th>
<th>DOB (both)</th>
<th>DOB</th>
<th>BldType</th>
<th>BldType</th>
<th>HIVPOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>'1974-01-30'</td>
<td>'1974-04-13'</td>
<td>'ANEG'</td>
<td>'ABPOS'</td>
<td>false</td>
</tr>
<tr>
<td>102</td>
<td>'1985-12-29'</td>
<td>'1986-03-12'</td>
<td>'APOS'</td>
<td>'ONEG'</td>
<td>true</td>
</tr>
<tr>
<td>103</td>
<td>'1979-03-29'</td>
<td>'1979-06-10'</td>
<td>'ABNEG'</td>
<td>'OPOS'</td>
<td>false</td>
</tr>
<tr>
<td>104</td>
<td>'1976-05-20'</td>
<td>'1976-08-01'</td>
<td>'APOS'</td>
<td>'ONEG'</td>
<td>true</td>
</tr>
<tr>
<td>105</td>
<td>'1987-08-25'</td>
<td>'1987-11-06'</td>
<td>'ABPOS'</td>
<td>'BNEG'</td>
<td>false</td>
</tr>
<tr>
<td>106</td>
<td>'1987-12-22'</td>
<td>'1988-03-04'</td>
<td>'APOS'</td>
<td>'ANEG'</td>
<td>true</td>
</tr>
<tr>
<td>107</td>
<td>'1968-06-20'</td>
<td>'1968-09-01'</td>
<td>'BNEG'</td>
<td>'APOS'</td>
<td>false</td>
</tr>
<tr>
<td>108</td>
<td>'1966-12-25'</td>
<td>'1967-03-08'</td>
<td>'BNEG'</td>
<td>'APOS'</td>
<td>true</td>
</tr>
<tr>
<td>109</td>
<td>'1985-11-17'</td>
<td>'1986-01-29'</td>
<td>'BNEG'</td>
<td>'APOS'</td>
<td>false</td>
</tr>
<tr>
<td>110</td>
<td>'1973-10-02'</td>
<td>'1973-12-14'</td>
<td>'ANEG'</td>
<td>'ABPOS'</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 7-2: Original and Level-1 Obfuscation Values for Patients Table

Observe in Table 7-2 how that partial knowledge of the original values could allow an attacker to reverse the obfuscation process. For instance, if one knew the original birthdates for patients 102 and 109, one might look at the obfuscated values and deduce that all birthdates were shifted by 73 days. If one knew the true HIV-positive values for patients 103, 104 and 110, one might deduce that the obfuscated HIVPOS values were simply the complements of the original values. Reversing the obfuscation could be done with the following SDDL table description:

```
<dynamic_table>
  <pool name="BTs">
    <choice name="APOS"/>
    <choice name="ANEG"/>
  </pool>
</dynamic_table>
```
7.1.2.2 Process Reversible Obfuscation (Level 2)

An obfuscation technique is process reversible if “[k]nowing the obfuscation technique or one of its standard processes can lead to complete or partial reversibility of the obfuscated data set.” [17]. Level 2 obfuscation usually takes the following form:

\[ \text{Value}_{\text{obfuscated}} = F(\text{Value}_{\text{original}}) \]

An attacker who knows the process associated with the function \( F() \) can construct the inverse function \( F'() \) and use it to compute the original values from the
obfuscated values. One typical way of enforcing Level 2 obfuscation is through the use of the random number generator, which can be considered to be a reversible function.

Level 2 obfuscation would typically be employed when there is a moderate threat of an attack, and the data’s originators desire that the data still be reversible. We could implement Level 2 obfuscation on all fields in the Patients table via the following SDDL table definition\(^\text{11}\):

\[
<\text{database}>
<\text{seed}71662741</\text{seed}>
<\text{pool} \text{name}="BTs"><\text{choice} \text{name}="APOS"/><\text{choice} \text{name}="ANEG"/><\text{choice} \text{name}="BPOS"/><\text{choice} \text{name}="BNEG"/><\text{choice} \text{name}="OPOS"/><\text{choice} \text{name}="ONEG"/><\text{choice} \text{name}="ABPOS"/><\text{choice} \text{name}="ABNEG"/></\text{pool}>

<\text{table} \text{name}="Sanitized2"><\text{variable} \text{name}="ORIG" \text{type}="\text{int}">
<\text{iteration}
\text{query}="\text{SELECT ID,DOB,BldType,HIVPOS FROM Patients}"/></\text{variable}>
<\text{field} \text{name}="ID" \text{type}="\text{int}">
<\text{formula}>0+\text{ORIG}$ID</\text{formula}>
</\text{field}>
<\text{field} \text{name}="DOB" \text{type}="\text{date}"
<\text{formula}>\text{DATE(ORIG}$DOB)+\text{IRND(20)}</\text{formula}>
</\text{field}>
<\text{field} \text{name}="BldType" \text{type}="\text{CHAR(5)}"
<\text{formula}>\text{BTs[ORIG}$BldType]+\text{IRND(8)}</\text{formula}>
</\text{field}>
<\text{field} \text{name}="HIVPOS" \text{type}="\text{bool}"
<\text{formula}>
\text{BOOL(ORIG}$\text{HIVPOS})!=(\text{IRND(2)==1})\)
</\text{formula}>
</\text{table}>
\]

\(^\text{11}\) The \text{IRND(x)} function returns a random integer between 0 and \(x-1\), inclusive.
In the table description above, the obfuscated DOB values are up to 19 days later than the original DOB values, the obfuscated BldType values are shifted up to 7 places from the original BldType values, and the obfuscated HIVPOS values are set to the original values exclusive-ored with a random binary variable. The obfuscation results are shown in Table 7-3.

<table>
<thead>
<tr>
<th>ID</th>
<th>DOB</th>
<th>BldType</th>
<th>HIVPOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>'1974-01-30'</td>
<td>'1974-02-13'</td>
<td>'ANEG'</td>
</tr>
<tr>
<td>102</td>
<td>'1985-12-29'</td>
<td>'1986-01-05'</td>
<td>'APOS'</td>
</tr>
<tr>
<td>103</td>
<td>'1979-03-29'</td>
<td>'1979-03-30'</td>
<td>'ABNEG'</td>
</tr>
<tr>
<td>104</td>
<td>'1976-05-20'</td>
<td>'1976-05-23'</td>
<td>'APOS'</td>
</tr>
<tr>
<td>105</td>
<td>'1987-08-25'</td>
<td>'1987-08-26'</td>
<td>'ABPOS'</td>
</tr>
<tr>
<td>106</td>
<td>'1987-12-22'</td>
<td>'1988-01-07'</td>
<td>'OPOS'</td>
</tr>
<tr>
<td>107</td>
<td>'1968-06-20'</td>
<td>'1968-07-08'</td>
<td>'BNEG'</td>
</tr>
<tr>
<td>108</td>
<td>'1966-12-25'</td>
<td>'1967-01-03'</td>
<td>'BNEG'</td>
</tr>
<tr>
<td>109</td>
<td>'1985-11-17'</td>
<td>'1985-11-18'</td>
<td>'BNEG'</td>
</tr>
<tr>
<td>110</td>
<td>'1973-10-02'</td>
<td>'1973-10-06'</td>
<td>'ANEG'</td>
</tr>
</tbody>
</table>

Table 7-3: Original and Level-2 Obfuscation Values for Patients Table

Note that there is no longer a one-to-one correlation between the original attribute values and the obfuscated attribute values. Because of this, Level 2 obfuscation is much harder to reverse than Level 1 obfuscation. In order to reverse Level 2 obfuscation (when obfuscation function \( F \) is a random number generator), the following must be known:

- The random number generator used during obfuscation
- The seed used for the random number generator
- The random stream algorithm used by the random number generator
- The exact sequence of random values generated

To ensure maximum difficulty in reversing Level-2 obfuscated data, Bakken advises the use of a sufficiently complex random number generator, as well as a non-trivial seed (stay away from 0 and 1) [17]. In our case, since we know that we generated the obfuscated data with SDG, it is relatively easy to provide an SDDL table description to restore the original data from the obfuscated data:

```xml
<database>
  <seed>71662741</seed>
  <pool name="BTs">
    <choice name="APOS"/>
    <choice name="ANEG"/>
    <choice name="BPOS"/>
    <choice name="BNEG"/>
    <choice name="OPOS"/>
    <choice name="ONEG"/>
    <choice name="ABPOS"/>
    <choice name="ABNEG"/>
  </pool>
</database>

<table name="Restored">
  <variable name="SANITIZED" type="int">
    <iteration>
      query="SELECT ID,DOB,BldType,HIVPOS FROM Sanitized2"
    </iteration>
  </variable>
  <field name="ID" type="int">
    <formula>0+ SANITIZED$ID</formula>
  </field>
  <field name="DOB" type="date">
    <formula>DATE(SANITIZED$DOB)-IRND(20)</formula>
  </field>
  <field name="BldType" type="CHAR(5)">
    <formula>
      BTs[SANITIZED$BldType]-IRND(8)
    </formula>
  </field>
  <field name="HIVPOS" type="bool">
    <formula>
      BOOL(SANITIZED$HIVPOS)!=(IRND(2)==1)
    </formula>
  </field>
</table>
```
7.1.2.3 Combination Reversible Obfuscation (Level 3)

To reverse-engineer this level of obfuscation, an attacker would need both knowledge of the obfuscation process and at least partial knowledge of the original data [17]. A Level 3 obfuscation formula might typically look something like this:

\[
\text{Value}_{\text{obfuscated}} = F(\text{Value}_{\text{original}}) + \text{offset}
\]

An attacker would need to know function \( F \) (and be able to compute inverse function \( F'() \)), and would also need some of the original values in order to deduce the correct value for an offset. As with Level 2 obfuscation, a random number generator is a commonly used function for implementing obfuscation.

Level 3 obfuscation would typically be employed when the threat of attack is moderate-to-high, and the data originators require that the obfuscated data still be reversible. Level 3 obfuscation of our Patients table might be effected through the following SDDL table definition:

```xml
<database>
  <seed>71662741</seed>
  <pool name="BTs">
    <choice name="APOS"/>
    <choice name="ANEG"/>
    <choice name="BPOS"/>
    <choice name="BNEG"/>
    <choice name="OPOS"/>
    <choice name="ONEG"/>
    <choice name="ABPOS"/>
    <choice name="ABNEG"/>
  </pool>

  <table name="Sanitized3">
    <variable name="ORIG" type="int">
```
<iteration
    query="SELECT ID,DOB,BldType,HIVPOS
    FROM Patients"/></iteration>

<field name="ID" type="int">
    <formula>0+ORIG$ID</formula>
</field>

<field name="DOB" type="date">
    <formula>DATE(ORIG$DOB)+IRND(21)-10</formula>
</field>

<field name="BldType" type="CHAR(5)">
    <formula>BTs[ORIG$BldType]+IRND(8)-4</formula>
</field>

<field name="HIVPOS" type="bool">
    <formula>BOOL(ORIG$HIVPOS)!(IRND(2)==1)</formula>
</field>

</table>
</database>

Note that the DOB and BldType field definitions now contain a function (IRND) and an offset. The HIVPOS definition is unchanged from the Level 2 obfuscation definition; it is difficult to define a meaningful difference between Level 2 and Level 3 obfuscation of a binary field. The results are shown in Table 7-4.

<table>
<thead>
<tr>
<th>ID (both)</th>
<th>DOB Orig.</th>
<th>DOB Obfusc.</th>
<th>BldType Orig.</th>
<th>BldType Obfusc.</th>
<th>HIVPOS Orig.</th>
<th>HIVPOS Obfusc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>'1974-01-30'</td>
<td>'1974-01-20'</td>
<td>'ANEG'</td>
<td>'BPOS'</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>102</td>
<td>'1985-12-29'</td>
<td>'1985-12-28'</td>
<td>'APOS'</td>
<td>'ABNEG'</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>103</td>
<td>'1979-03-29'</td>
<td>'1979-03-27'</td>
<td>'ABNEG'</td>
<td>'ABPOS'</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>104</td>
<td>'1976-05-20'</td>
<td>'1976-05-14'</td>
<td>'APOS'</td>
<td>'BPOS'</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>105</td>
<td>'1987-08-25'</td>
<td>'1987-08-25'</td>
<td>'ABPOS'</td>
<td>'ANEG'</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>106</td>
<td>'1987-12-22'</td>
<td>'1987-12-24'</td>
<td>'OPOS'</td>
<td>'BPOS'</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>107</td>
<td>'1968-06-20'</td>
<td>'1968-06-30'</td>
<td>'BNEG'</td>
<td>'ANEG'</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>108</td>
<td>'1966-12-25'</td>
<td>'1966-12-27'</td>
<td>'BNEG'</td>
<td>'APOS'</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>109</td>
<td>'1985-11-17'</td>
<td>'1985-11-21'</td>
<td>'BNEG'</td>
<td>'ABNEG'</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>110</td>
<td>'1973-10-02'</td>
<td>'1973-09-24'</td>
<td>'ANEG'</td>
<td>'ANEG'</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 7-4: Original and Level-3 Obfuscation Values for Patients Table

Our Level 3 obfuscation can be reversed by the following SDDL table definition:

<database>
<seed>71662741</seed>
<pool name="BTs">
  <choice name="APOS"/>
  <choice name="ANEG"/>
  <choice name="BPOS"/>
  <choice name="BNEG"/>
  <choice name="OPOS"/>
  <choice name="ONEG"/>
  <choice name="ABPOS"/>
  <choice name="ABNEG"/>
</pool>

<table name="Restored">
  <variable name="SANITIZED" type="int">
    <iteration query="SELECT ID,DOB,BldType,HIVPOS FROM Sanitized3"/>
  </variable>
  <field name="ID" type="int">
    <formula>0+ SANITIZED$ID</formula>
  </field>
  <field name="DOB" type="date">
    <formula>
      DATE(SANITIZED$DOB) - IRND(21) + 10
    </formula>
  </field>
  <field name="BldType" type="CHAR(5)">
    <formula>
      BTs[SANITIZED$BldType] - IRND(8) + 4
    </formula>
  </field>
  <field name="HIVPOS" type="bool">
    <formula>
      BOOL(SANITIZED$HIVPOS) != (IRND(2) == 1)
    </formula>
  </field>
</table>
</database>

7.1.3 Irreversible Obfuscation (Levels 4-6)

Where reversible obfuscation techniques were based on original values, irreversible obfuscation techniques are not. Thus, it is not possible to reverse-engineer individual original values from the obfuscated data set. However, depending upon the
level of irreversible obfuscation employed, an attacker might still be able to infer some of the statistical characteristics of the original data set from the obfuscated data set.

7.1.3.1 Statistically Accurate or Dependency Preserving Irreversible Obfuscation (Level 4)

Level 4 obfuscation is irreversible in terms of reproducing the original values, yet still preserves information about the original data set in the form of statistical relationships and/or relational dependencies\textsuperscript{12}. Such obfuscation might preserve the following types of information:

- The mean of a set of values
- A statistical distribution of a set of values
- A functional dependency of one field upon another field
- Join selectivity constraints
- Foreign key dependencies

This type of obfuscation is common in situations where the data originators would like to give a third party a “realistic” data set without giving them the real data, and where data obfuscation reversibility is not required. We could enforce Level 4 obfuscation on all fields in our Patients table with the following SDDL table definition:

\begin{verbatim}
<database>
  <pool name="BTs">
    <choice name="APOS"><weight>2</weight></choice>
  </pool>
</database>
\end{verbatim}

\textsuperscript{12} The concept of “statistically accurate” is a generalization of Bakken’s concept of “mean-preserving” [17].
<choice name="ANEG"><weight>2</weight></choice>
<choice name="BNEG"><weight>3</weight></choice>
<choice name="OPOS"><weight>1</weight></choice>
<choice name="ABPOS"><weight>1</weight></choice>
<choice name="ABNEG"><weight>1</weight></choice>

<table name="Sanitized4">
  <field name="ID" type="int">
    <iteration base="101" count="10"/>
  </field>
  <field name="DOB" type="date">
    <dist>
      <tier prob="0.2" min="1960-01-01" max="1969-12-31"/>
      <tier prob="0.4" min="1970-01-01" max="1979-12-31"/>
      <tier prob="0.4" min="1980-01-01" max="1989-12-31"/>
    </dist>
  </field>
  <field name="BldType" type="CHAR(5)">
    <formula>BTs</formula>
  </field>
  <field name="HIVPOS" type="bool">
    <dist>
      <tier prob="0.6" min="false" max="false"/>
      <tier prob="0.4" min="true" max="true"/>
    </dist>
  </field>
</table>

Note that each obfuscated field produces values that have some statistical significance. The DOB field values are constrained to be distributed according to the observed distribution of the original values. The BldType values come from a pool that is weighted according to the observed BldType distribution of the original data. The HIVPOS field values are constrained to be false 60% of the time and true 40% of the time, as was observed in the original data. Note that we could have used queries to calculate these statistics of the original data. The results are shown in Table 7-5.
<table>
<thead>
<tr>
<th>ID</th>
<th>DOB</th>
<th>BldType</th>
<th>HIVPOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>'1974-01-30'</td>
<td>'1984-05-28'</td>
<td>'ANEG'</td>
</tr>
<tr>
<td>102</td>
<td>'1985-12-29'</td>
<td>'1988-03-02'</td>
<td>'APOS'</td>
</tr>
<tr>
<td>103</td>
<td>'1979-03-29'</td>
<td>'1970-01-14'</td>
<td>'ABNEG'</td>
</tr>
<tr>
<td>104</td>
<td>'1976-05-20'</td>
<td>'1983-02-07'</td>
<td>'APOS'</td>
</tr>
<tr>
<td>105</td>
<td>'1987-08-25'</td>
<td>'1980-08-02'</td>
<td>'ABPOS'</td>
</tr>
<tr>
<td>106</td>
<td>'1987-12-22'</td>
<td>'1970-02-14'</td>
<td>'APOS'</td>
</tr>
<tr>
<td>107</td>
<td>'1968-06-20'</td>
<td>'1964-05-11'</td>
<td>'BNEG'</td>
</tr>
<tr>
<td>108</td>
<td>'1966-12-25'</td>
<td>'1971-08-10'</td>
<td>'BNEG'</td>
</tr>
<tr>
<td>109</td>
<td>'1985-11-17'</td>
<td>'1975-05-22'</td>
<td>'BNEG'</td>
</tr>
<tr>
<td>110</td>
<td>'1973-10-02'</td>
<td>'1976-07-05'</td>
<td>'ANEG'</td>
</tr>
</tbody>
</table>

Table 7-5: Original and Level-4 Obfuscation Values for Patients Table

We observe that it is hard to enforce exact distribution constraints in a small data set. For example, our obfuscated HIVPOS values were 50% true and 50% false, instead of the 40/60 constraint was specified for that field. If the generated data set was larger (say, 1000 patients), then the specified constraints would be more evident in the generated data.

It can be observed without loss of generality that join selectivity between Table A and Table B (where Table A contains a foreign key reference to Table B) can be preserved by maintaining the cardinalities of both Table A and Table B. In a more general sense, the join selectivity between the two tables can be preserved by maintaining the ratio between $|A|$ and $|B|$.

To preserve foreign key dependencies, foreign key fields should be generated via SDDL QueryPools or QueryIterations to assure that the foreign key field contains a valid value from the key field that it references. Foreign key fields should not undergo independent obfuscation.
For example, if field $F_1$ from table $A$ references field $F_2$ of table $B$, then table $A$ depends upon table $B$, and table $B$ should be generated first. The SDDL generation constraint for $A.F_1$ should look like this:

```
<table name="A">
  [... other fields ...]
  <field name="F1">
    <queryPool query="SELECT F2 from B"/>
    OR
    <iteration query="SELECT F2 from B"/>
  </field>
  [... other fields ...]
</table>
```

The referenced field ($B.F_2$ in this example) can be obfuscated in any fashion that the user prefers, so long as all obfuscated $F_2$ values are unique. As long as referencing fields ($A.F_1$ in this example) use a QueryIteration or QueryPool referencing the foreign field as their generation constraint, then referential integrity will be preserved.

### 7.1.3.2 Domain Preserving Irreversible Obfuscation (Level 5)

Level 5 obfuscation preserves the general domain for an attribute. For a continuous attribute, Level 5 obfuscation might preserve the range of the attribute values. For a discrete field, Level 5 obfuscation would preserve the discrete values associated with the range of a field. Level 5 obfuscation would typically be used on a field that was of secondary importance to a third party, but might still be used in realistic queries used to test the obfuscated data set. The Patients table could be obfuscated at Level 5 as follows:

```
<database>
  <pool name="BTs">
    <choice name="APOS"/>
    <choice name="ANEG"/>
  </pool>
</database>
```
<pool>

<table name="Sanitized5">

<table>
<thead>
<tr>
<th>ID</th>
<th>DOB</th>
<th>BldType</th>
<th>HIVPOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>'1974-01-30'</td>
<td>'ANEG'</td>
<td>false</td>
</tr>
<tr>
<td>102</td>
<td>'1985-12-29'</td>
<td>'APOS'</td>
<td>true</td>
</tr>
<tr>
<td>103</td>
<td>'1979-03-29'</td>
<td>'ABNEG'</td>
<td>false</td>
</tr>
<tr>
<td>104</td>
<td>'1976-05-20'</td>
<td>'APOS'</td>
<td>true</td>
</tr>
<tr>
<td>105</td>
<td>'1987-08-25'</td>
<td>'ABPOS'</td>
<td>false</td>
</tr>
<tr>
<td>106</td>
<td>'1987-12-22'</td>
<td>'OPOS'</td>
<td>true</td>
</tr>
<tr>
<td>107</td>
<td>'1968-06-20'</td>
<td>'BNEG'</td>
<td>false</td>
</tr>
<tr>
<td>108</td>
<td>'1966-12-25'</td>
<td>'BNEG'</td>
<td>false</td>
</tr>
<tr>
<td>109</td>
<td>'1985-11-17'</td>
<td>'BNEG'</td>
<td>false</td>
</tr>
</tbody>
</table>

Note that the field values are constrained to be in the correct domain and range, but carry no other statistical significance. The DOB values are simply constrained to be between 1960 and 1990, the BldType values are in the right domain but are unweighted, and the HIVPOS values are just random Boolean variables. The results are shown in Table 7-6.
7.1.3.3 Opaque Irreversible Obfuscation (Level 6)

Level 6 is the highest level of obfuscation; it renders the original values and statistical relationships of the original data completely opaque. Level 6 is not even required to preserve field value domains. There are a few general techniques for Level 6 obfuscation:

- Output a constant value for a field.
- Output completely random values for a field.
- Omit the field entirely in the sanitized table. This is schema obfuscation, and will be discussed in the next section.

Level 6 obfuscation is typically employed on a field that is needed as a “placeholder”, but will not be used in any significant test queries. Level 6 obfuscation for our Patients table might be achieved with the following SDDL table description:

```xml
<database>
    <table name="Sanitized6">
        <field name="ID" type="int">
            <iteration base="101" count="10"/>
        </field>
        <field name="DOB" type="date">
            <min>#1800-01-01#</min>
            <max>#2006-01-01#</max>
        </field>
        <field name="BldType" type="CHAR(5)">
            <formula>"type"+IRND(10)</formula>
        </field>
        <field name="HIVPOS" type="bool">
            <formula>false</formula>
        </field>
    </table>
</database>
```
The obfuscated DOB values are valid dates, but the range of these values has been vastly increased from the range of the original values. The obfuscated BldType values are no longer valid blood types, just generic typeX values. Finally, the HIVPOS values have all been set to false. Table 7-7 shows the results.

<table>
<thead>
<tr>
<th>ID</th>
<th>DOB Orig.</th>
<th>DOB Obfusc.</th>
<th>BldType Orig.</th>
<th>BldType Obfusc.</th>
<th>HIVPOS Orig.</th>
<th>HIVPOS Obfusc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>'1975-01-30'</td>
<td>'1934-01-20'</td>
<td>'ANEG'</td>
<td>'type7'</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>102</td>
<td>'1985-12-29'</td>
<td>'1829-02-15'</td>
<td>'APOS'</td>
<td>'type9'</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>103</td>
<td>'1979-03-29'</td>
<td>'1993-05-28'</td>
<td>'ABNEG'</td>
<td>'type4'</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>104</td>
<td>'1976-05-20'</td>
<td>'1871-06-02'</td>
<td>'APOS'</td>
<td>'type8'</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>105</td>
<td>'1978-08-25'</td>
<td>'1953-01-21'</td>
<td>'ABNEG'</td>
<td>'type9'</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>106</td>
<td>'1978-12-22'</td>
<td>'1883-09-10'</td>
<td>'OPOS'</td>
<td>'type8'</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>107</td>
<td>'1968-06-20'</td>
<td>'1962-07-21'</td>
<td>'BNEG'</td>
<td>'type7'</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>108</td>
<td>'1966-12-25'</td>
<td>'1910-03-11'</td>
<td>'BNEG'</td>
<td>'type5'</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>109</td>
<td>'1985-11-17'</td>
<td>'2003-01-14'</td>
<td>'BNEG'</td>
<td>'type4'</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 7-7: Original and Level-6 Obfuscation Values for Patients Table

7.2 Other Types of Obfuscation

In addition to data obfuscation, there are other ways to obfuscate the contents of a table. A table’s schema may be obfuscated, and a table’s cardinality may be obfuscated. For this section, all examples will assume the Employees table shown in Table 7-8 below.

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>LastName</th>
<th>FirstName</th>
<th>DOB Orig.</th>
<th>DOB Obfusc.</th>
<th>SSN Orig.</th>
<th>SSN Obfusc.</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>'Macdonald'</td>
<td>'Nicholas'</td>
<td>'1955-02-21'</td>
<td>'453-81-8761'</td>
<td>'1934-01-20'</td>
<td>'453-81-8761'</td>
<td>78719</td>
</tr>
<tr>
<td>102</td>
<td>'Campbell'</td>
<td>'Daniel'</td>
<td>'1979-09-29'</td>
<td>'791-78-7500'</td>
<td>'1934-01-20'</td>
<td>'791-78-7500'</td>
<td>67569</td>
</tr>
<tr>
<td>103</td>
<td>'Young'</td>
<td>'Anthony'</td>
<td>'1980-04-09'</td>
<td>'519-75-6681'</td>
<td>'1934-01-20'</td>
<td>'519-75-6681'</td>
<td>87517</td>
</tr>
<tr>
<td>104</td>
<td>'Clark'</td>
<td>'Madison'</td>
<td>'1982-07-01'</td>
<td>'529-94-4916'</td>
<td>'1934-01-20'</td>
<td>'529-94-4916'</td>
<td>98671</td>
</tr>
<tr>
<td>105</td>
<td>'Clark'</td>
<td>'Christopher'</td>
<td>'1963-06-04'</td>
<td>'644-41-9813'</td>
<td>'1934-01-20'</td>
<td>'644-41-9813'</td>
<td>73196</td>
</tr>
<tr>
<td>106</td>
<td>'Stewart'</td>
<td>'Hannah'</td>
<td>'1971-09-07'</td>
<td>'842-61-3037'</td>
<td>'1934-01-20'</td>
<td>'842-61-3037'</td>
<td>106946</td>
</tr>
</tbody>
</table>

Table 7-8: Reference Employees Table
7.2.1 Schema Obfuscation

Schema obfuscation involves the alteration of the schema for the purpose of blurring or hiding the meaning of the data.

7.2.1.1 Attribute Suppression

If an attribute’s data values are confidential, and/or if that attribute simply is not needed by the third party receiving the obfuscated table, then it is not necessary to include that attribute in the obfuscated table. Suppose that we judged the LastName and FirstName attributes of our Employees table as too sensitive to pass on. We could simply drop these fields from the obfuscated version of the table, as shown in Table 7-9.

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>DOB</th>
<th>SSN</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>'1955-02-21'</td>
<td>'453-81-8761'</td>
<td>78719</td>
</tr>
<tr>
<td>102</td>
<td>'1979-09-29'</td>
<td>'791-78-7500'</td>
<td>67569</td>
</tr>
<tr>
<td>103</td>
<td>'1980-04-09'</td>
<td>'519-75-6681'</td>
<td>87517</td>
</tr>
<tr>
<td>104</td>
<td>'1982-07-01'</td>
<td>'529-93-4916'</td>
<td>98671</td>
</tr>
<tr>
<td>105</td>
<td>'1963-06-04'</td>
<td>'644-41-9813'</td>
<td>73196</td>
</tr>
<tr>
<td>106</td>
<td>'1971-09-07'</td>
<td>'842-61-3037'</td>
<td>106946</td>
</tr>
</tbody>
</table>

Table 7-9: Attribute-Suppressed Employees Table

7.2.1.2 Generic Attribute Naming

One way to alter a schema to blur the meaning of data is to change the attribute names from something meaningful to something generic. Suppose that we changed the attribute names in Table 7-9 as follows:

<table>
<thead>
<tr>
<th>F1</th>
<th>F1</th>
<th>F3</th>
<th>F4</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>'1955-02-21'</td>
<td>'453-81-8761'</td>
<td>78719</td>
</tr>
<tr>
<td>102</td>
<td>'1979-09-29'</td>
<td>'791-78-7500'</td>
<td>67569</td>
</tr>
<tr>
<td>103</td>
<td>'1980-04-09'</td>
<td>'519-75-6681'</td>
<td>87517</td>
</tr>
<tr>
<td>104</td>
<td>'1982-07-01'</td>
<td>'529-93-4916'</td>
<td>98671</td>
</tr>
</tbody>
</table>
The meanings of the EmployeeID and Salary fields from the original table are now obscured. The DOB field values are still recognizable as dates, but not immediately as a date-of-birth. The SSN field values still look like social security numbers.

### 7.2.1.3 Attribute Decomposition

The meanings of some classes of attributes are easily recognizable by their format. Phone numbers, dates and social security numbers fall into this category. If such fields are decomposed into their component parts, their collective meanings will have been obscured for an attacker. For example, suppose that we re-formatted the Employees table from Table 7-10 as follows:

<table>
<thead>
<tr>
<th></th>
<th>F1</th>
<th>F1</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1955</td>
<td>2</td>
<td>21</td>
<td>453</td>
<td>81</td>
<td>8761</td>
<td>78719</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>1979</td>
<td>9</td>
<td>29</td>
<td>791</td>
<td>78</td>
<td>7500</td>
<td>67569</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>1980</td>
<td>4</td>
<td>9</td>
<td>519</td>
<td>75</td>
<td>6681</td>
<td>87517</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>1982</td>
<td>7</td>
<td>1</td>
<td>529</td>
<td>93</td>
<td>4916</td>
<td>98671</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>1963</td>
<td>6</td>
<td>4</td>
<td>644</td>
<td>41</td>
<td>9813</td>
<td>73196</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>1971</td>
<td>9</td>
<td>7</td>
<td>842</td>
<td>61</td>
<td>3037</td>
<td>106946</td>
<td></td>
</tr>
</tbody>
</table>

Table 7-11: Attribute-Suppressed Employees Table with Generic Attribute Names and Decomposed Attributes

The DOB and SSN field values are now not easily recognizable. To make the meaning even more obscure, the order of the attribute columns could be shuffled.
7.2.1.4 Attribute Insertion

To further obfuscate the meaning of a table, one could insert extraneous attributes with random values. Consider the effect of introducing arbitrarily valued attributes into Table 7-11. The resulting table might look something like Table 7-12, below. Fields F3 and F7 (values are shaded) have been inserted into the table. An attacker might suspect that one of the fields is the “Salary” field, but is it F7 or F10? Likewise, an attacker might suspect that a date is encoded into the table, but which fields comprise the date: F2/F3/F4, F2/F3/F5 or F2/F4/F5?

<table>
<thead>
<tr>
<th></th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>F9</th>
<th>F10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101</td>
<td>1955</td>
<td>5</td>
<td>2</td>
<td>21</td>
<td>453</td>
<td>108029</td>
<td>81</td>
<td>8761</td>
<td>78719</td>
</tr>
<tr>
<td>2</td>
<td>102</td>
<td>1979</td>
<td>4</td>
<td>9</td>
<td>29</td>
<td>791</td>
<td>60308</td>
<td>78</td>
<td>7500</td>
<td>67569</td>
</tr>
<tr>
<td>3</td>
<td>103</td>
<td>1980</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>519</td>
<td>104231</td>
<td>75</td>
<td>6681</td>
<td>87517</td>
</tr>
<tr>
<td>4</td>
<td>104</td>
<td>1982</td>
<td>10</td>
<td>7</td>
<td>1</td>
<td>529</td>
<td>73121</td>
<td>93</td>
<td>4916</td>
<td>98671</td>
</tr>
<tr>
<td>5</td>
<td>105</td>
<td>1963</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>644</td>
<td>66987</td>
<td>41</td>
<td>9813</td>
<td>73196</td>
</tr>
<tr>
<td>6</td>
<td>106</td>
<td>1971</td>
<td>1</td>
<td>9</td>
<td>7</td>
<td>842</td>
<td>84642</td>
<td>61</td>
<td>3037</td>
<td>106946</td>
</tr>
</tbody>
</table>

Table 7-12: Attribute-Suppressed Employees Table with Generic Attribute Names, Decomposed Attributes and Extraneous Fields

7.2.2 Cardinality Obfuscation

It is sometimes the case that the actual number of rows in a table might be considered sensitive information, and thus would be a candidate for obfuscation. In this case, it makes sense to either add “fake” data rows to (or delete rows of data from) the obfuscated data set.

Suppose that we wanted to randomly delete half of the rows in our reference Employees table. The results are shown in Table 7-13.
Table 7-13: Attribute-Suppressed Employees Table with Generic Attribute Names, Decomposed Attributes, Extraneous Fields and Rows Suppressed

In this example, the randomly generated values for the $F_3$ and $F_7$ fields changed because the random streaming sequence was perturbed.

### 7.2.3 SDDL Support for Schema and Cardinality Obfuscation

The following SDDL table definition was used to produce the data in Table 7-13 from the original Employees table:

```xml
<database>
  <load funcName="decompose"/>
  <table name="emp2">
    <variable name="ORIG" type="int">
      <iteration
        query="SELECT EmployeeID, DOB, SSN, Salary FROM EMPLOYEES">
        <repeatDist>
          <tier prob="0.75" min="0" max="0"/>
          <tier prob="0.25" min="1" max="1"/>
        </repeatDist>
      </iteration>
    </variable>
    <field name="F1" type="int">
      <formula>0+ORIG$EmployeeID</formula>
    </field>
    <field name="F2" type="int">
      <formula>0+decompose(ORIG$DOB,"-",0)</formula>
    </field>
    <field name="F3" type="int">
      <formula>1+IRND(12)</formula>
    </field>
    <field name="F4" type="int">
      <formula>0+decompose(ORIG$DOB,"-",1)</formula>
    </field>
    <field name="F5" type="int">
      <formula>0+decompose(ORIG$DOB,"-",1)</formula>
    </field>
  </table>
</database>
```
<field name="F1" type="int">
  <formula>0+decompose(ORIG$DOB,"-",2)</formula>
</field>
<field name="F2" type="int">
  <formula>0+decompose(ORIG$DOB,"-",0)</formula>
</field>
<field name="F3" type="int">
  <formula>60000+IRND(50000)</formula>
</field>
<field name="F4" type="int">
  <formula>0+decompose(ORIG$SSN,"-",1)</formula>
</field>
<field name="F5" type="int">
  <formula>0+decompose(ORIG$SSN,"-",2)</formula>
</field>
<field name="F6" type="int">
  <formula>0+decompose(ORIG$SSN,"-",0)</formula>
</field>
<field name="F7" type="int">
  <formula>0+decompose(ORIG$SSN,"-",1)</formula>
</field>
<field name="F8" type="int">
  <formula>0+decompose(ORIG$SSN,"-",2)</formula>
</field>
<field name="F9" type="int">
  <formula>0+decompose(ORIG$SSN,"-",1)</formula>
</field>
<field name="F10" type="int">
  <formula>0+ORIG$Salary</formula>
</field>
</table>
</database>

Note the following about the table definition above:

- The LastName and FirstName fields from the original Employees table were completely ignored, thereby enforcing attribute suppression for those two attributes.
- The fields are generically named “F1” through “F10”, thereby enforcing generic attribute naming.
- The plug-in “decompose” function was used to decompose the DOB and SSN attributes into less meaningful integer values.
- The F3 and F7 fields are inserted as extraneous random-valued fields to further obscure the meaning of the data.
- The repeat distribution associated with the ORIG field iteration caused rows to be arbitrarily dropped from the obfuscated table. For this small
sample size, a 75% “miss” rate had to be specified to suppress half of the rows in the table.

### 7.3 A Warning about Data Inference

The data inference problem can be summarized as follows: data items that one might classify as “not sensitive” can sometimes be combined together and with external data to infer the value of data items that one considers sensitive. Many papers have been written about data inference, of which [16] is a good representative.

As an example, consider the following table:

<table>
<thead>
<tr>
<th>LastName</th>
<th>FirstName</th>
<th>University</th>
<th>Degree</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Bob</td>
<td>BYU</td>
<td>BS</td>
<td>65449</td>
</tr>
<tr>
<td>Smith</td>
<td>James</td>
<td>UCLA</td>
<td>MS</td>
<td>80500</td>
</tr>
<tr>
<td>Brown</td>
<td>Joe</td>
<td>CMU</td>
<td>BS</td>
<td>71200</td>
</tr>
<tr>
<td>Anderson</td>
<td>Will</td>
<td>Texas</td>
<td>BA</td>
<td>57800</td>
</tr>
<tr>
<td>Green</td>
<td>Jack</td>
<td>Stanford</td>
<td>PhD</td>
<td>94500</td>
</tr>
</tbody>
</table>

Table 7-14: Sample Employees Table

We might consider the combination of LastName, FirstName, and Salary to be sensitive information. Thus, we might release the following table to a third party for an analysis of the relation of salary to school and degree:

<table>
<thead>
<tr>
<th>University</th>
<th>Degree</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYU</td>
<td>BS</td>
<td>65449</td>
</tr>
<tr>
<td>UCLA</td>
<td>MS</td>
<td>80500</td>
</tr>
<tr>
<td>CMU</td>
<td>BS</td>
<td>71200</td>
</tr>
<tr>
<td>Texas</td>
<td>BA</td>
<td>57800</td>
</tr>
<tr>
<td>Stanford</td>
<td>PhD</td>
<td>94500</td>
</tr>
</tbody>
</table>

Table 7-15: Sanitized Employees Table

However, if an attacker were to obtain this data, and that attacker had some knowledge of the university and degree associated with certain employees, then the
attacker could infer the first and last names associated with certain salaries. If, for example, the attacker knew that Bob Jones had a BS from BYU, then the attacker could infer that Bob Jones had a salary of 65449. In this fashion, data considered non-sensitive (university and degree), combined with external knowledge (where Bob Jones went to school), could be used to infer data considered sensitive (Bob Jones’ salary).

Unfortunately, there is no way for SDG (or any other program) to automatically detect which data is sensitive and which is not. Nor is there a way to detect that external knowledge might render sensitive data items inferable. The moral of the story is that great care must be taken when obfuscating data inorder to prevent the inference of sensitive data. In particular, one must be careful about any fields one chooses to leave unobfuscated (i.e., Level 0 obfuscation). These unobfuscated, (seemingly) “safe” fields can support the inference of sensitive fields.

7.4 Summary

In this chapter, various methods of data obfuscation were presented. Reversible methods of obfuscation are based on original individual data values, and so an attacker with knowledge of the obfuscation method might be able to reproduce the original data. Irreversible methods of obfuscation are not based on original individual data values, and so original data values cannot be deduced by an attacker; however, statistical trends of the original data might still be preserved. Finally, other methods of table obfuscation were presented.
All types of data and table obfuscation presented herein are enforceable via SDDL. Thus, the SDG framework can be an effective tool for the exportation of obfuscated data sets.
8 APPLICATION: GENERATING TEN YEARS OF STORE/ITEM/SALES DATA

This chapter chronicles the use of the SDG framework to generate 10 years of store/item/sales data for a major retailer. This application of SDG was important because it proved that SDG is scalable and can produce a very large amount of data relatively quickly by taking advantage of parallelism.

8.1 Schema for Sales Table

Our task for this application was to generate 10 years worth of information of the Sales table for the retailer. The schema shown for the Sales table is given in Table 8-1.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StoreNum</td>
<td>DECIMAL(09,05)</td>
<td>Store ID</td>
</tr>
<tr>
<td>ItemNum</td>
<td>INTEGER</td>
<td>Item ID</td>
</tr>
<tr>
<td>Week</td>
<td>INTEGER</td>
<td>Week ID (includes year)</td>
</tr>
<tr>
<td>SellPrice</td>
<td>DECIMAL(09,02)</td>
<td>Sale price of item</td>
</tr>
<tr>
<td>WklyQty</td>
<td>INTEGER</td>
<td>Sum of daily quantities</td>
</tr>
<tr>
<td>SatQty</td>
<td>INTEGER</td>
<td>Quantity sold on Saturday</td>
</tr>
<tr>
<td>SunQty</td>
<td>INTEGER</td>
<td>Quantity sold on Sunday</td>
</tr>
<tr>
<td>MonQty</td>
<td>INTEGER</td>
<td>Quantity sold on Monday</td>
</tr>
<tr>
<td>TueQty</td>
<td>INTEGER</td>
<td>Quantity sold on Tuesday</td>
</tr>
<tr>
<td>WedQty</td>
<td>INTEGER</td>
<td>Quantity sold on Wednesday</td>
</tr>
<tr>
<td>ThuQty</td>
<td>INTEGER</td>
<td>Quantity sold on Thursday</td>
</tr>
<tr>
<td>FriQty</td>
<td>INTEGER</td>
<td>Quantity sold on Friday</td>
</tr>
<tr>
<td>WklySales</td>
<td>DECIMAL(09,02)</td>
<td>WklyQty * SellPrice</td>
</tr>
</tbody>
</table>

Table 8-1: Schema for Sales Table
The unique key for the table is composed of \((\text{StoreNum, ItemNum, SellPrice})\). There can be multiple entries in the table for any given \((\text{StoreNum, ItemNum})\) pair, within a given week, if the SellPrice field for that \((\text{StoreNum, ItemNum})\) pair changes during that week.

The Week field is an integer encoded as:

\[(\text{Year} - 1900) * 100 + \text{week}\]

In other words, the tenth fiscal week of 1997 would result in a Week attribute value of \(9710\). The sixteenth fiscal week of 2005 would be encoded as \(10516\).

### 8.2 Generation Constraints

The following generation constraints were specified for the Sales table:

- There are 140,000,000 entries per week in the Sales table
- There are 2,892 unique store IDs
- There are 737,253 unique item IDs
- The number of unique items sold per week by each store is distributed according to Table 8-2. For example, 460 stores sell between 30,001 and 40,000 unique items each week.
- The number of times an item appears in the Sales table (each week) is distributed according to Table 8-3. For example, 419636 items make between 2 and 100 appearances per week in the Sales table.
- Beyond that, we were free to enforce constraints of our choosing on the data.
<table>
<thead>
<tr>
<th>Range Minimum</th>
<th>Range Maximum</th>
<th>Number of Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10,000</td>
<td>7</td>
</tr>
<tr>
<td>10,001</td>
<td>20,000</td>
<td>180</td>
</tr>
<tr>
<td>20,001</td>
<td>30,000</td>
<td>280</td>
</tr>
<tr>
<td>30,001</td>
<td>40,000</td>
<td>460</td>
</tr>
<tr>
<td>40,001</td>
<td>50,000</td>
<td>643</td>
</tr>
<tr>
<td>50,001</td>
<td>60,000</td>
<td>496</td>
</tr>
<tr>
<td>60,001</td>
<td>70,000</td>
<td>490</td>
</tr>
<tr>
<td>70,001</td>
<td>80,000</td>
<td>315</td>
</tr>
<tr>
<td>80,001</td>
<td>90,000</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 8-2: Unique Items Sold per Store

<table>
<thead>
<tr>
<th>Range Minimum</th>
<th>Range Maximum</th>
<th>Number of Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>131263</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>419636</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>46317</td>
</tr>
<tr>
<td>200</td>
<td>300</td>
<td>26845</td>
</tr>
<tr>
<td>300</td>
<td>400</td>
<td>18620</td>
</tr>
<tr>
<td>400</td>
<td>500</td>
<td>13602</td>
</tr>
<tr>
<td>500</td>
<td>600</td>
<td>10724</td>
</tr>
<tr>
<td>600</td>
<td>700</td>
<td>8544</td>
</tr>
<tr>
<td>700</td>
<td>800</td>
<td>6992</td>
</tr>
<tr>
<td>800</td>
<td>900</td>
<td>6010</td>
</tr>
<tr>
<td>900</td>
<td>1000</td>
<td>5211</td>
</tr>
<tr>
<td>1000</td>
<td>1100</td>
<td>4641</td>
</tr>
<tr>
<td>1100</td>
<td>1200</td>
<td>4520</td>
</tr>
<tr>
<td>1200</td>
<td>1300</td>
<td>4477</td>
</tr>
<tr>
<td>1300</td>
<td>1400</td>
<td>4362</td>
</tr>
<tr>
<td>1400</td>
<td>1500</td>
<td>2906</td>
</tr>
<tr>
<td>1500</td>
<td>1600</td>
<td>2564</td>
</tr>
<tr>
<td>1600</td>
<td>1700</td>
<td>2135</td>
</tr>
<tr>
<td>1700</td>
<td>1800</td>
<td>1892</td>
</tr>
<tr>
<td>1800</td>
<td>1900</td>
<td>1687</td>
</tr>
<tr>
<td>1900</td>
<td>2000</td>
<td>1523</td>
</tr>
<tr>
<td>2000</td>
<td>2100</td>
<td>1428</td>
</tr>
<tr>
<td>2100</td>
<td>2200</td>
<td>1339</td>
</tr>
<tr>
<td>2200</td>
<td>2300</td>
<td>1289</td>
</tr>
<tr>
<td>2300</td>
<td>2400</td>
<td>1250</td>
</tr>
<tr>
<td>2400</td>
<td>2500</td>
<td>1243</td>
</tr>
<tr>
<td>2500</td>
<td>2600</td>
<td>1176</td>
</tr>
<tr>
<td>2600</td>
<td>2700</td>
<td>1166</td>
</tr>
<tr>
<td>2700</td>
<td>2800</td>
<td>1123</td>
</tr>
</tbody>
</table>
Table 8-3: Table Frequency per Item

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2800</td>
<td>2900</td>
<td>905</td>
</tr>
<tr>
<td>2900</td>
<td>10000</td>
<td>1847</td>
</tr>
</tbody>
</table>

8.3 Enforcing Constraints with SDDL

8.3.1 The item_dist Pool

Before covering the SDDL definition for the Sales table, the item_dist pool needs to be explained. For each row generated in the Sales table, an item number (e.g., 1-737253) must be generated, and it must be distributed according to Table 8-3. The item_dist pool is used to enforce this distribution.

Each row in Table 8-3 contains a frequency range and the number of items that sold according to that frequency range. For simplicity’s sake, it was assumed that all items in a frequency range were numbered consecutively, though in real life that may not be the case. Thus, items 1-131263 make one appearance per week in the Sales table, items 131264-550899 make between 2 and 100 appearances per week in the Sales table, and so on.

If one assumes that the average frequency of each row in the table is the midpoint between the lower and upper frequency boundaries, one can then construct a weighted pool as follows:

```xml
<pool name="item_dist">
  <choice name="1">
    <minItem>1</minItem>
    <maxItem>131263</maxItem>
    <weight>131263</weight>
  </choice>
  <choice name="2-100">
    <minItem>131264</minItem>
    <maxItem>550899</maxItem>
  </choice>
</pool>
```
In the pool above, the weight of each element is calculated as

\[(\text{Range}_{\text{min}} + \text{Range}_{\text{max}})/2\)*NumRangeItems\]

In this manner, the weights enforce the distribution described in Table 8-3. Each time an item number is generated, a weighted choice in the item_dist pool will be selected, and the item number will be taken from the range specified by the \(<\text{minItem}>\) and \(<\text{maxItem}>\) values associated with that choice.

However, there is one small problem: The weights add up to considerably more than 140,000,000, which is the total number of rows generated in the Sales table each week. This means that the entire range of the item_dist pool will not be represented in each weekly data generation, which means that the constraints specified by Table 8-3 will not be preserved.

In order to preserve the weight total of 140,000,000, the formula for each weight was changed to

\[((\text{Range}_{\text{min}} + \text{Range}_{\text{max}})/2)-16\)*NumRangeItems\]

In other words, we shifted the average frequency of each row to 16 below the midpoint of the frequency range. By so doing, the weights added up correctly and the
constraints specified by Table 8-3 were preserved. The final item_dist pool definition looked like this:

```xml
<pool name="item_dist">
  <choice name="1">
    <minItem>1</minItem>
    <maxItem>131263</maxItem>
    <weight>131263</weight>
  </choice>
  <choice name="2-100">
    <minItem>131264</minItem>
    <maxItem>550899</maxItem>
    <weight>14267624</weight>
  </choice>
  <choice name="101-200">
    <minItem>550900</minItem>
    <maxItem>597216</maxItem>
    <weight>6206478</weight>
  </choice>
  [...]
  <choice name="2801-2900">
    <minItem>734486</minItem>
    <maxItem>735390</maxItem>
    <weight>2564770</weight>
  </choice>
  <choice name="2900-">
    <minItem>735391</minItem>
    <maxItem>737237</maxItem>
    <!--Assumed avg weight of 3100--> 
    <weight>5725700</weight>
  </choice>
</pool>
```

The first and last choices in the pool are treated differently, because their frequency ranges do not follow the normal rules.

### 8.3.2 The Sales Table

The description of a year’s worth of Sales table data begins with the following preamble:
As we wish to generate 52 weeks’ worth of data, our outer iteration will be defined as follows:

```
<variable name="week_count" type="INTEGER">
   <iteration base="0" count="52"/>
</variable>
```

The next inner iteration will iterate through all stores for the week:

```
<field name="StoreNum" type="DECIMAL(09,05)">
   <iteration base="1" count="2892"/>
</field>
```

The `item_rows` variable implements the constraints found in Table 8-2 for the store in question by calculating the number of rows that will be generated for the current store in the current week:

```
<variable name="item_rows" type="int">
   <dist>
      <tier prob="0.0024" min="0" max="10000"/>
      <tier prob="0.0622" min="10001" max="20000"/>
      <tier prob="0.0968" min="20001" max="30000"/>
      <tier prob="0.1591" min="30001" max="40000"/>
      <tier prob="0.2224" min="40001" max="50000"/>
      <tier prob="0.1715" min="50001" max="60000"/>
      <tier prob="0.1694" min="60001" max="70000"/>
      <tier prob="0.1089" min="70001" max="80000"/>
      <tier prob="0.0073" min="80001" max="90000"/>
   </dist>
</variable>
```

This iteration causes `item_rows` rows to be generated with current `StoreNum`:

```
<variable name="item_entry" type="int">
   <iteration base="1" count="item_rows"/>
</variable>
```
These fields generate an ItemNum value for this row, using the item_dist pool:

```xml
<variable name="dist_class" type="string">
    <formula>item_dist</formula>
</variable>

[field name="ItemNum" type="int">
    <min>item_dist[dist_class].minItem</min>
    <max>item_dist[dist_class].maxItem</max>
</field>
```

Next, the Week field is described for this row. Here, we are assuming that the year is 2006, but that assumption changed with each year of data generated:

```xml
[field name="Week" type="int">
    <formula>10600+week_count</formula>
</field>
```

Next, the SellPrice field is described. Although no constraints were specified by the customer for this field, some were specified by the author.

```xml
[field name="SellPrice" type="DECIMAL(09,02)">
    <dist>
        <tier prob="0.8" min="0.10" max="20" step="0.01"/>
        <tier prob="0.15" min="20" max="150" step="0.01"/>
        <tier prob="0.05" min="150" max="2500" step="0.01"/>
    </dist>
</field>
```

The following field defines the WklyQty field value as anywhere between 1 and 100 (inclusive):

```xml
[field name="WklyQty" type="int">
    <min>1</min>
    <max>100</max>
</field>
```
The following generate the daily quantities for this row, making sure that they add up to the \texttt{WklyQty} field value:

\begin{verbatim}
<field name="SatQty" type="int">
  <min>0</min>
  <max>WklyQty/4</max>
</field>
<field name="SunQty" type="int">
  <min>0</min>
  <max>WklyQty/4</max>
</field>
<field name="MonQty" type="int">
  <min>0</min>
  <max>WklyQty/4</max>
</field>
<field name="TueQty" type="int">
  <min>0</min>
  <max>WklyQty/4</max>
</field>
<field name="WedQty" type="int">
  <min>0</min>
  <max>WklyQty-(SatQty+SunQty+MonQty+TueQty)</max>
</field>
<field name="ThuQty" type="int">
  <min>0</min>
  <max>WklyQty-(SatQty+SunQty+MonQty+TueQty+WedQty)</max>
</field>
<field name="FriQty" type="int">
  <formula>
    WklyQty-(SatQty+SunQty+MonQty+TueQty+WedQty+ThuQty)
  </formula>
</field>

Finally, the \texttt{WklySales} value for the row is calculated and the table definition is ended.

\begin{verbatim}
<field name="WklySales" type="DECIMAL(09,02)">
  <formula>WklyQty*SellPrice</formula>
</field>
</table>
\end{verbatim}
The table definition above will result in one year of Sales data being generated, and the data will conform to all of the constraints specified by the user. To generate multiple years of data, the definition above can replicated. The replicated files must all have different random seeds, and the Week field must be defined based upon the desired generation year.

8.4 Generation Results

The SDDL definition in 8.3.2 was used to generate ten years’ worth of Sales table data. The data was generated one year at a time. Note that the outer loop in the Sales data description is 52 elements long (one element for each week of a year). Since SDG parallelizes across the outer iteration elements of a table, this meant that a year’s worth of data could be evenly split among 52, 26, 13, 4, 2, or 1 generation processes.

Data was generated on two separate hardware grid configurations. The description and results for each generation are described in this section.

8.4.1 HP Enterprise Technology Center (ETC), Atlanta, GA

The first hardware in Atlanta was a cluster of 4 HP rx4640s, each containing 4 1.6-GHz Itanium processors and 12 GB of RAM, for a total of 16 available processors. It was determined that each year would be generated simultaneously across 13 generation processes. Due to limited disk space, it was decided that only 2 years of “Sales” table data would be generated at this site.

It took about 4 hours, on average, to generate each year of “Sales” table data. This means that, on average, SDG was using parallelism to generate a little over 500,000 rows per second in this environment.
8.4.2 HP Performance Center, Cupertino, CA

The hardware in Cupertino was an HP rx8640, which contained 16 1.6-GHz Itanium processors and 128 GB of main memory. It was again decided that the data generation would be split among 13 processes, each generation process producing 4 weeks for a given year. This hardware contained sufficient disk space for SDG to generate ten years’ worth of Sales table data.

It took about 5.5 hours, on average, to generate each year of “Sales” table data. This works out to about 367,000 rows per second. A total of 10 years (4.2 TB) of data was generated. The total time taken to generate the 10 years of data was a little over two days, which included some cluster reboots.

Given nearly identical hardware, why did SDG produce data at a slower rate in Cupertino than it had done in Atlanta? Unfortunately, we only had access to the hardware for a short time, and were never able to find a definitive answer to that question. A number of theories were proposed:

- Having all 13 generation processes on the same physical machine resulted in more memory contention than having them spread out over 4 smaller machines.
- The operating system associated with the Atlanta machines did a better job at keeping a generation process tied to a physical processor, which minimized the time taken up with process-swapping.
- The Java JVM on the Cupertino system was set up less optimally than the Java JVM on the Atlanta system.
8.5 Summary

This application of SDG was important for a number of reasons:

- It showed that SDG could effectively utilize cluster computing to generate large amounts of synthetic data in parallel. The generator reached generation speeds of over 500,000 rows per second, and generated a total of 4.2 Terabytes of data.

- The scalability, speed, and volume of data vindicated our decision to write the SDG framework in Java. The SDG framework was easily moved from platform to platform without recompilation.

- This application showed that nebulous constraints such as those in Table 8-3 could be effectively modeled in SDDL.
9 APPLICATION: GENERATING THE SEMANTICALLY COMPLEX HALLUX APPLICATION

Chapter 8 showcased the ability to scale the SDG framework to generate a several-billion-row table by employing parallel generation processes on a high-end computing cluster. It is also desirable to scale SDG in another dimension, that of generating a complex data set comprised of many interrelated tables.

The Enterprise Systems project at the University of Arkansas Walton College of Business hosts a collection of very large databases that can be used by academic institutions (see http://enterprise.waltoncollege.uark.edu/systems.asp?show=TUN). This project conceived the idea of a fictitious music industry database, to be used for educational purposes. This fictitious database would center around a company called “Hallux Productions”, and would track such music-industry data as artists, agents, albums, performances and sales. Our SDG project was approached about generating synthetic data for such a database; this chapter describes the approach taken in doing so.

The Hallux database poses problems typical of most tabular synthetic data generation applications: many interrelated tables and application-specific business rules. That being the case, the techniques described in this chapter could be applied to a wide class of synthetic data generation problems.

9.1 Description of Hallux Database

Figure 9-1 shows an E-R diagram of the entire Hallux database. This diagram imparts a general sense of the number of tables in the database as well as the relationships among the tables. For a complete and detailed description of the Hallux database, see
Appendix C, which contains a full description of the database schema, the business rules associated with each table, and the SDDL code used to generate each table.

9.2 Logical Challenges Encountered

This section describes some of the logical decisions and techniques that went into generating the Hallux database.

9.2.1 Order of Table Generation

When generating multiple related tables, some order must be imposed on the sequence in which tables are generated. If table B is dependent upon table A, then table A must be generated first, then table B. In general, the “leaf” tables (i.e., those without dependencies on other tables) should be generated first, then those tables that depend upon the “leaf” tables, and so on so that dependencies are honored throughout the generation of the database.

It can be difficult to identify dependencies. For example, in Figure 9-1, does the Person table depend upon the Band_Member table, which depends upon the Band table? That may be one logical way to look at it: first generate Band, then generate Band_Member from Band, then generate Person from Band_Member. However, Band has a hidden dependency on Person: if a band name involves a proper name (i.e., “Crazy Larry”), then one of the band members should typically have the name of “Larry”. Therefore, Person was generated before Band, so that band names involving proper names could have access to the names of the band members.

The lesson here is that sometimes the proper table ordering cannot be determined until one fully understands the business rules associated with each table. Once one learns
that band names can include proper names, for example, then the proper names must be
generated first.

Figure 9-1: Hallux Database E-R Diagram
After carefully analyzing all dependencies, the Hallux tables were generated in the following order:

1. **Instrument** – generated from “Instruments” pool.
2. **Genre** – generated from “GenreInstrument” pool.
3. **Zip_Code** – generated from “StateZip” and “ZipLatLon” pools.
4. **State** – generated from “StateAbbrToName” pool.
5. **Venue** – generated from “StateZip”, “names” and “streetnames” pools.
6. **Producer** – generated from internal “ProducerNames” pool.
7. **Order_Source** – generated from internal “sources” pool.
8. **Person** – generated from “StateZip”, “names”, and “streetnames” pools.
9. **Agent** – dependency on Person table.
10. **Band** – dependencies on Person, Agent tables.
11. **Contract** – dependencies on Band, Agent tables.
13. **Band_Member** – dependencies on Person, Band tables.
14. **Member_Instrument** – dependencies on Band_Member, Band_Genre tables.
15. **Album** – dependency on Band table.
16. **Song** – dependency on Album table.
17. **Video** – generated from external and internal pools.
18. **ItemType** – generated from “ItemTypes” pool.
19. **Item** – dependencies on Album, Band, Song, Video tables.
20. **Customer** – generated from external and internal pools.

22. **Order_Header** – dependencies on Order_Detail, Order_Source tables.

23. **Performance** – dependencies on Album, Venue, Agent tables.

24. **Customer_Profile** – dependencies on Customer, Album, Song, Band, Order_Detail tables.


### 9.2.2 Band/Album/Song Naming

It is often the case that a table attribute will require an English-language phrase as its value; special pools and formulas can be employed to construct these phrases. Such was the case for the **Band.band_name**, **Album.album_name** and **Song.song_name** attributes in the Hallux database. In order for the music database to be realistic, the band, album and song names needed to be realistic. It would have been undesirable, for example, to simply name the albums “album1, album2, …”. Also, it was important that names did not repeat too often, and that they were more or less grammatically correct.

To provide this type of reality, an SDDL file called “Words.xml” was constructed. This file contains the following pools:

- “Verbs”: A pool of 78 English verbs.
• “Adjectives”: A pool of 253 English adjectives.

• “Adverbs”: A pool of 172 English adverbs.

• “Nouns”: A pool of 67 English nouns, in both singular and plural form.

• “Determiners”: A pool consisting of “The”, “Their”, “His”, “Her”, “My”, “Our”, and “Your”.

• “Prepositions”: A pool consisting of “Over”, “Under”, “Through”, “By”, “Below”, “Above” and “Against”.


In addition, we already had a pool of common first and last names in several ethnicities, as well as a pool of U.S. city and state names. These were also used in the naming process.

The rules for producing band names are shown in Table 9-1. In the case where a band was actually an individual (i.e., there was 1 member in the band), the top-most rule was always used. Since this is a relatively small table, and because in reality band names are typically unique, the goal here was complete uniqueness among band names. Accordingly, out of the 400 band names produced, there were no duplicates.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Firstname&gt;+” “+&lt;Lastname&gt;</code></td>
<td>“Michael Clark”</td>
</tr>
<tr>
<td><code>&lt;Verb&gt;+” The” “+&lt;SingularNoun&gt;</code></td>
<td>“Beat The Wheel”</td>
</tr>
<tr>
<td>“The “+&lt;SingularNoun&gt;+” “+&lt;Grouptype&gt;`</td>
<td>“The Salvation Disaster”</td>
</tr>
<tr>
<td>“The “+&lt;Lastname&gt;+” “+&lt;Grouptype&gt;`</td>
<td>“The Herrera Group”</td>
</tr>
<tr>
<td><code>&lt;Firstname&gt;+” And The “+&lt;PluralNoun&gt;</code></td>
<td>“Suman And The Sages”</td>
</tr>
</tbody>
</table>
The rules for producing album names are shown in Table 9-2. The relative weights for the fourth and fifth rules were lowered to cut down on repeat naming. One goal here was to have a high degree of diversity in album naming; in the real world, it is rare (but not impossible) for albums by two groups to have the same name. Out of 2641 album names produced, 2589 were unique. This was determined to be a satisfactory degree of uniqueness among album names.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Relative Weight</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;SingularNoun&gt;</code>+” “+&lt;SingularNoun&gt;</td>
<td>14</td>
<td>“Shakespeare Generation”</td>
</tr>
<tr>
<td><code>&lt;Verb&gt;</code>+” “+&lt;PluralNoun&gt;</td>
<td>14</td>
<td>“Forbid Fools”</td>
</tr>
<tr>
<td><code>&lt;Adjective&gt;</code>+” “+&lt;PluralNoun&gt;</td>
<td>14</td>
<td>“Glamorous Oceans”</td>
</tr>
<tr>
<td><code>&lt;Verb&gt;</code></td>
<td>1</td>
<td>“Spill”</td>
</tr>
<tr>
<td><code>&lt;Adverb&gt;</code></td>
<td>1</td>
<td>“Beautifully”</td>
</tr>
<tr>
<td><code>&lt;Adjective&gt;</code>+” “+&lt;SingularNoun&gt;</td>
<td>14</td>
<td>“Magical Universe”</td>
</tr>
<tr>
<td><code>&lt;Preposition&gt;</code>+” “+&lt;Determiner&gt;+” “+&lt;Adjective&gt;+” “+&lt;SingularNoun&gt;</td>
<td>14</td>
<td>“Through Their Green Ambition”</td>
</tr>
<tr>
<td><code>&lt;Adverb&gt;</code>+” “+&lt;Adjective&gt;</td>
<td>14</td>
<td>“Abusively Fortunate”</td>
</tr>
<tr>
<td><code>&lt;Verb&gt;</code>+” “+&lt;Adverb&gt;</td>
<td>14</td>
<td>“Amplify Dramatically”</td>
</tr>
</tbody>
</table>

Table 9-2: Album Naming Rules

The rules for producing song names are shown in Table 9-3. A `<Placename>` has a 90% chance of being a city name and a 10% chance of being a state name. One goal here was to have a high degree of diversity in song naming; in the real world, it is rare (but not impossible) for songs by two groups to have the same name. Of the 27936 song
names generated, 26606 were unique; this was determined to be a satisfactory degree of uniqueness among song names.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Adverb&gt;++&lt;Adjective&gt;</td>
<td>“Toughly Magical”</td>
</tr>
<tr>
<td>&lt;Verb&gt;++&lt;Determiner&gt;++&lt;SingularNoun&gt;</td>
<td>“Forbid Her Exaltation”</td>
</tr>
<tr>
<td>&lt;Verb&gt;++&lt;Adverb&gt;</td>
<td>“Condemn Brightly”</td>
</tr>
<tr>
<td>&lt;Determiner&gt;++&lt;Adjective&gt;++&lt;SingularNoun&gt;</td>
<td>“The Knowing Loss”</td>
</tr>
<tr>
<td>&lt;Placename&gt;++&lt;Adverb&gt;</td>
<td>“Buckeye Watchfully”</td>
</tr>
<tr>
<td>&lt;Adjective&gt;++&lt;Placename&gt;</td>
<td>“Magical In Wyoming”</td>
</tr>
<tr>
<td>&lt;Adjective&gt;++&lt;Firstname&gt;</td>
<td>“Jumbled Jacob”</td>
</tr>
<tr>
<td>&lt;Adjective&gt;++&lt;PluralNoun&gt;</td>
<td>“Fearless Grounds”</td>
</tr>
<tr>
<td>&lt;Adjective&gt;++And++&lt;Adjective&gt;</td>
<td>“Craven And Giddy”</td>
</tr>
</tbody>
</table>

Table 9-3: Song Naming Rules

Note also the following:

- While the number of repeated song and album names was within tolerance, it could be further reduced by additional rules and also by additional entries in the various “Words” pools. Both of these actions would increase the diversity of the respective name spaces.

- Similar techniques were used to produce the venue names in the Venue table, as well as the corporate customer names in the Customer table.

9.2.3 Determinism Issues

The Hallux data set was produced in “test” mode to a “MySQL” database, and was produced in “production” mode to a Teradata database. The “test” mode was used for intermediate testing, and the “production” mode was used to generate the Hallux database to its final destination. It was disturbing to note that the MySQL data set was
not identical to the Teradata data set. After some investigation, the cause of the problem became clear.

In a MySQL database, query data will (by default) be returned in the order that the data was inserted into the database. In a Teradata database, that does not hold true – the Teradata query engine will return data in “jumbled” order. This is not in any way an error on the part of the Teradata; the relational model does not require any order to a relation. Rather, it illustrates what can happen when “lazy” unordered queries are employed to collect data. An example is in order.

Suppose that you created a table with the following SDDL description:

```xml
<table name="foo">
    <field name="F1" type="int">
        <iteration base="1013" count="3"/>
    </field>
</table>
```

If you generated that table to a MySQL database, then submitted a query “select F1 from foo”, you would get the results back in the order “1013, 1014, 1015”. If you generated that table to a Teradata database and then performed the same query, the results could come back in any order, like “1014, 1013, 1015”.

Taking the example one step further, suppose you then created a table with the following SDDL description:

```xml
<pool name="colors">
    <choice name="red"/>
    <choice name="green"/>
    <choice name="blue"/>
</pool>
<table name="bar">
    <field name="foovalue" type="int">
        <iteration query="select F1 from foo"/>
    </field>
</table>
```
The color column will be generated in a deterministic order, depending upon the random seed. For our purposes, say the color column is generated as “blue”, “red”, “green”. Then, on the MySQL database, the bar table is generated as follows:

<table>
<thead>
<tr>
<th>foovalue</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1013</td>
<td>blue</td>
</tr>
<tr>
<td>1014</td>
<td>red</td>
</tr>
<tr>
<td>1015</td>
<td>green</td>
</tr>
</tbody>
</table>

Table 9-4: Sample Generation to MySQL

However, on a Teradata database, the foovalue query could return data in any order, possibly resulting in something like this:

<table>
<thead>
<tr>
<th>foovalue</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1014</td>
<td>blue</td>
</tr>
<tr>
<td>1013</td>
<td>red</td>
</tr>
<tr>
<td>1015</td>
<td>green</td>
</tr>
</tbody>
</table>

Table 9-5: Sample Generation to Teradata

The two tables are obviously different, even though the random number sequencing was the same, because the ordering of the query results used in the foovalue field was different. This type of problem was causing the data generated to the Teradata to differ from the data generated to the MySQL. The solution was to specify explicit ordering in every query issued during data generation. In our small example,
determinism could be preserved by changing the iteration query for the \texttt{foovalue} field to (bold face added by author):

\begin{verbatim}
<iteration query="select F1 from foo order by F1"/>
\end{verbatim}

Once this explicit ordering was enforced on every query issued during generation, the Hallux data sets generated to the MySQL and Teradata platforms were identical.

\section*{9.2.4 Differences in SQL Interpreters}

There were instances when the MySQL SQL interpreter would accept an INSERT statement, and the Teradata SQL interpreter would reject the same INSERT statement. For example, Java has some rounding issues when converting floating-point numbers to strings, and an INSERT statement produced by SDG might look something like this:

\begin{verbatim}
INSERT INTO Contract
    (Band_Id,Contract_Id,Agent_Id,Begin_Date,End_Date,Album_Rev_Pct,Album_Count,Live_Rev_Pct,Live_Count) VALUES
    (1161,11610,200012,null,null,0.060000000000000005,4,0.41000000000000003,null)
\end{verbatim}

For MySQL, this was acceptable. The Teradata, however, threw a numeric overflow exception when confronted with that statement. The solution\textsuperscript{13} was to change

\textsuperscript{13}This solution fit the needs of the Hallux data set, and most other data sets. The ultimate solution, though, would be to round according to the type given for them field. For example, a field of type “NUMERIC(9,4)” would result in rounding to 4 decimal places. The author has added this fix to his to-do list.
SDG to always print real numbers out to six decimal places, changing our sample

INSERT statement to:

```sql
INSERT INTO Contract
(Band_Id, Contract_Id, Agent_Id, Begin_Date, End_Date, Album_Rev_Pct, Album_Count, Live_Rev_Pct, Live_Count) VALUES
(1161, 11610, 200012, null, null, 0.06000, 4, 0.410000, null)
```

The INSERT statement above was handled correctly by both SQL interpreters.

Another issue with interpreters was the handling of embedded apostrophes in quoted values in INSERT statements. The following statement is acceptable in MySQL:

```sql
INSERT INTO Venue
(Venue_Id, Venue_Name, Street_Address, Zip_Code, Zip_Code_Ext, Contact_Phone) VALUES
(1000, "Gang's Club", "362 West Clinton Expressway", "37887", ",", "8752417320")
```

However, the statement above is rejected by the Teradata\(^{14}\), because quoted fields must be quoted by single-quotes. For the Teradata (and most other SQL interpreters), embedded apostrophes are represented by two single-quotes in a row. The sample statement above could therefore be changed to:

```sql
INSERT INTO Venue
(Venue_Id, Venue_Name, Street_Address, Zip_Code, Zip_Code_Ext, Contact_Phone) VALUES
(1000, 'Gang''s Club', '362 West Clinton Expressway', '37887', '', '8752417320')
```

This way of dealing with embedded apostrophes was acceptable to both MySQL and Teradata.

\(^{14}\) This section is not meant to be a criticism of Teradata SQL, but rather to point out the differences possible between various SQL interpreters.
Finally, different SQL interpreters provide support for different operators, and interpret operators differently. For example, in MySQL SQL, the ‘/’ operator means “floating point divide”, the “div” operator means “integer divide”, and a floor() function is supported. In Teradata SQL, the ‘/’ can mean either “integer divide” or “floating point divide” depending upon the operands, and no floor() function is supported. The result is that some queries that involved division had to be written differently for MySQL than they were for the Teradata.

9.2.5 Using Temporary Attributes

There were times when a needed attribute was missing from the formal definition of a table, and so one was temporarily added in order to generate a dependent table. A good example of this is the Customer_Profile table.

Customer_Profile contains a Gender field to signify whether the customer is male or female, and Customer_Profile is supposed to be derived from data in Customer. However, gender is not an attribute of the Customer table, so there is really no way to know the gender of the entries in the Customer table. To deal with this, a temporary Gender attribute is generated for the individual customers in the Customer table; each first name in the names table has a gender associated with it, so this information is easy to output for each row. After the Customer_Profile table has completed generation, the Gender field of Customer can be deleted.

Temporary attributes are also useful for resolving circular data dependencies. If a dependent table contains an attribute needed by a source table, then that attribute can be generated in the source table and copied to the dependent table. This was done with the
Order_Detail table. In order to properly form an order, it is important to know whether the customer is an individual or a corporation. Unfortunately, the Order_Detail table does not contain a Customer_ID field. So, a temporary temp_custid field was generated each order in the Order_Detail table, then copied to the Order_Header table.

When generation is complete, the data is organized exactly as specified (i.e., no Gender attribute in the Customer table, and no temp_custid field in the Order_Detail table). However, during generation, some temporary fields are generated for later use and then deleted when their purpose has been fulfilled.

9.2.6 Ensuring Uniqueness in Key Values

Assigning key values for multiple interrelated tables was a challenge. In the real world, key values assigned to albums, songs and artists are independent. In the synthetic generation world, there are reasons to assign these values more systematically:

- It is difficult to assign random key values and guarantee that they will be unique.
- Relying on an “auto increment” mechanism to assign consecutive key values can result in non-deterministic value assignments when data is generated in parallel.
- It is sometimes convenient to encode dependencies or extra data into key values.
The approach to avoiding randomly generated duplicates within a domain involved use of formulas to systematically generate the next member in a deterministic pattern that avoided duplicates. For instance, the key value formulas for several Hallux tables are shown in Table 9-6.

<table>
<thead>
<tr>
<th>Key</th>
<th>Formula</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Band.band_id</td>
<td>Numbers 1000-1399</td>
<td>400 bands</td>
</tr>
<tr>
<td>Person.person_id (band member)</td>
<td>100000 + (band_id*10) + member index</td>
<td>At most 6 members per band. Range: 110000-113995.</td>
</tr>
<tr>
<td>Album.album_id</td>
<td>band_id * 20 + album index</td>
<td>At most 11 albums per band. Range: 20000-27982.</td>
</tr>
<tr>
<td>Song.song_id</td>
<td>album_id * 20 + song index</td>
<td>At most 15 songs per album. Range: 400001-559649.</td>
</tr>
<tr>
<td>Video.video_id</td>
<td>Numbers 50000-50199</td>
<td>200 Videos.</td>
</tr>
<tr>
<td>Performance.performance_id</td>
<td>album_id * 100 + performance index</td>
<td>At most 40 performances per album release</td>
</tr>
<tr>
<td>Order_Detail.order_id</td>
<td>1000000 + order_day*100 + order index</td>
<td>At most 50 orders per day (avg). Range: 1000001-1620853.</td>
</tr>
<tr>
<td>Item.item_id</td>
<td>album_id</td>
<td>song_id</td>
</tr>
</tbody>
</table>

Table 9-6: Selected Hallux Key Value Relationships

These methods for calculating key values assured that the key values were unique, and also encoded important data into several keys. For example, we know that song_id X comes from album_id floor(X/20). We know that album_id Y was produced by band_id floor(Y/20). And we know that order_id 1000001 was the first order received on day number 0, or January 1, 1990 (Hallux inception date).
9.2.7 Using Scalable SQL Operations

It is sometimes the case that SQL operations that are reasonably fast with small
data sets are prohibitively slow when applied to large data sets. In other words, the
operations do not scale well. One good example of this was found in the
Order_Detail table description.

Originally, candidate songs and albums for an order line were chosen as follows:

```xml
<variable name="songPool" type="int">
    <queryPool query="select song_id from Album, Song
        where Album.album_id = Song.album_id and
        Album.release_date &lt;= '[orderdate]'
        order by song_id"/>
</variable>
<variable name="albumId" type="int">
    <formula>songPool/20</formula>
</variable>
```

In other words, a song was chosen from all of the albums that had a
release_date before the order date. Then, the corresponding album ID was
calculated by dividing the candidate song_id by 20. This method worked fine for
small data sets. However, when the sizes of the Album and Song tables became very
large, it became very expensive to join the Album and Song tables every time that a
candidate song was needed.

To increase scalability, the sequence was changed as follows:

```xml
<variable name="albumPool" type="int">
    <queryPool query="select album_id from Album
        where release_date &lt;= '[orderdate]'
        order by album_id"/>
</variable>
<variable name="songPool" type="int">
    <queryPool query="select song_id from Song
        where album_id = [albumPool]
        order by song_id"/>
</variable>
```
There is no longer a join required; instead, the join is broken into two queries. First, a candidate \textit{album\_id} is identified, and then a random \textit{song\_id} from that album is chosen. This modification, along with the fact that an index on \textit{song\_id} was created in the \texttt{Song} table, resulted in a significant speedup in generating the \texttt{Order\_Detail} table.

9.3 Generating the Data

When generating large amounts of data to a database, one would typically like to generate the data to a text file (or set of text files), and then load the text file into the database using a fast-load application associated with the database. Experience has shown that it is usually much faster to load the data from a text file than it is to generate the data directly to a data base via JDBC [34].

However, because the Hallux database had so many interdependent tables, it was decided that the data would be generated directly to the database. The two-step method of generating to a file, then fast-loading the file to a database, would have resulted in a sequence of “generate table 1”, “load table 1”, “generate table 2”, “load table 2”, and so on. Since the author had no way of fast-loading data to the Teradata from the command line, the two-step method would have required user interaction after each of 25 tables was generated. Instead, all 25 tables were generated directly to the Teradata, allowing all 25 tables to be generated from a single batch file. Essentially, generation speed was traded for the convenience of “fire and forget” data generation.
There were two platforms used in the generation of the Hallux data: a “Test” platform to debug the SDG specification and a “Production” platform to download the final data to the target DBMS. The characteristics of these platforms are summarized in Table 9-7 below.

<table>
<thead>
<tr>
<th></th>
<th>“Test” Platform</th>
<th>“Production” Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Ubuntu Linux</td>
<td>Windows Server 2003 R2</td>
</tr>
<tr>
<td>Processor</td>
<td>2.4 GHz Intel Celeron</td>
<td>3.06 GHz Intel Xeon</td>
</tr>
<tr>
<td>Main Memory</td>
<td>512 MB</td>
<td>11.7 GB</td>
</tr>
<tr>
<td>Database</td>
<td>(Local) MySQL 5.0</td>
<td>(Remote) Teradata</td>
</tr>
<tr>
<td>Dedicated</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 9-7: Hallux Generation Platform Descriptions

Surprisingly, the data was generated much more quickly on the “Test” platform than on the “Production” platform. The results are summarized in Table 9-8. Despite the fact that the “Production” platform clearly had better hardware, the fact that the “Test” platform generated to a local database made the generation on that platform much faster. The reason for the difference was that SQL INSERT statements are performed one at a time during data generation; for large tables, the network latency time for each INSERT statement resulted in a significant time penalty for the “Production” platform with its remote database.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Table Generation Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>“Test” Platform</td>
</tr>
<tr>
<td>Instrument</td>
<td>0.5</td>
</tr>
<tr>
<td>Genre</td>
<td>0.5</td>
</tr>
<tr>
<td>Zip_Code</td>
<td>31.7</td>
</tr>
<tr>
<td>State</td>
<td>0.6</td>
</tr>
<tr>
<td>Venue</td>
<td>0.7</td>
</tr>
<tr>
<td>Producer</td>
<td>0.5</td>
</tr>
<tr>
<td>Item</td>
<td>Time 1</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Order_Source</td>
<td>0.5</td>
</tr>
<tr>
<td>Person</td>
<td>3.1</td>
</tr>
<tr>
<td>Agent</td>
<td>0.7</td>
</tr>
<tr>
<td>Band</td>
<td>3.0</td>
</tr>
<tr>
<td>Contract</td>
<td>2.7</td>
</tr>
<tr>
<td>Band_Genre</td>
<td>1.3</td>
</tr>
<tr>
<td>Band_Member</td>
<td>3.5</td>
</tr>
<tr>
<td>Member_Instrument</td>
<td>3.5</td>
</tr>
<tr>
<td>Album</td>
<td>5.3</td>
</tr>
<tr>
<td>Song</td>
<td>23.4</td>
</tr>
<tr>
<td>Video</td>
<td>0.8</td>
</tr>
<tr>
<td>ItemType</td>
<td>0.5</td>
</tr>
<tr>
<td>Item</td>
<td>30.8</td>
</tr>
<tr>
<td>Customer</td>
<td>6.8</td>
</tr>
<tr>
<td>Order_Detail</td>
<td>1132.6</td>
</tr>
<tr>
<td>Order_Header</td>
<td>338.6</td>
</tr>
<tr>
<td>Performance</td>
<td>69.8</td>
</tr>
<tr>
<td>Customer_Profile</td>
<td>26.2</td>
</tr>
<tr>
<td>Customer_Genre</td>
<td>13.1</td>
</tr>
<tr>
<td><strong>Total Generation Time</strong></td>
<td><strong>0H, 29M, 34S</strong></td>
</tr>
</tbody>
</table>

Table 9-8: Table Generation Times for Hallux DB

The time penalty associated with remote database generation resulted in experimentation with the use of `PreparedStatement` instead of line-at-a-time `INSERT` operations. Theoretically, the use of `PreparedStatement` for repetitive `INSERT` operations saves time in two ways: the statements are pre-parsed, and a number of `INSERT` statements can be batched in one `PreparedStatement`, which should reduce the effect of network latency.

The `PreparedStatement` method resulted in noticeable speedup for tables which generated large batches of uninterrupted `INSERT` statements, like `Song` and `Customer`. However, it resulted in only marginal speedup at best for tables with `SELECTs` intermixed with the `INSERTs`, such as `Order_Detail`. Thus, generation
of the entire Hallux database showed only a marginal speedup when PreparedStatements were employed during generation.

More importantly, it appears that the Teradata does not fully support PreparedStatements. Specifically, Teradata did not handle NULL values in PreparedStatements in some cases. As portability has a higher priority than a marginal speedup, it was decided that PreparedStatements would not replace line-at-a-time INSERT statements in the SDG generation engine.

9.4 Summary

Chapter 8 dealt with the generation of a single large table, and showcased the speed with which such a table could be generated when employing parallel generation processes. This chapter described a more typical situation, where a moderate number (25) of small-to-medium-sized inter-related tables, some with specific business logic associated with them, were described and generated.

A number of lessons were learned that are useful in generating this type of data set:

- Tables must be generated in order according to dependencies; if table A depends upon table B, then table B must be generated first. A careful analysis of all dependencies – both “hard” dependencies from an E-R diagram and “soft” dependencies associated with business rules – can be very helpful in ordering the tables correctly the first time.
- Methods of generating realistic, diverse English-language phrases were developed.
• In order to maintain determinism across multiple DBMS platforms, all SQL queries issued during data generation should be explicitly ordered by means of an “ORDER BY” clause.

• Missing information and circular data dependencies can be overcome by generating (and then later deleting) extra fields in tables.

• Carefully constructed key values can fulfill uniqueness requirements and can also contain encoded relational data.

• It is important that SQL queries issued during data generation scale well to large data sets.

• When generating data directly to a database, there is a significant time penalty for generating to a remote database as opposed to a local database. If possible, always generate the data locally.
In previous chapters, techniques for generating tabular data were demonstrated. Could one also generate data outside the tabular domain? Could one generate, for example, a synthetic resume? Or a synthetic scientific paper? Could one synthetically generate random (but valid) source code for a computer program? All of these are textual, linguistic applications of synthetic data generation, as opposed to tabular applications.

The automatic generation of textual documents is not without precedent. The author wrote a C program in the early 1990s that was effective at synthetically generating quarterly employee reports, to relieve him of that burden. The framework in [8] was capable of synthetically generating XML files. Perhaps most humorously, a group of MIT students recently submitted a synthetically generated paper, with the confusing title of “Rooter: a methodology for the typical unification of access points and redundancy”, to a conference [35]. The paper was grammatically correct, but made no sense whatsoever. It was simply an arbitrary collection of popular computer science phrases, buzzwords and graphs. Nevertheless, it was initially accepted by the conference! However, these are all examples of special-purpose generation frameworks producing textual data.

This chapter demonstrates the use of a general-purpose synthetic data generator (SDG) to generate legal strings for context-free languages (including regular languages). Grammars can be modeled as pools, and SDDL generation definitions can be constructed
to process those pools. The point of this chapter is to show how SDG can generate textual data that conforms to a user-defined set of rules.

10.1 Definition of a Context-Free Grammar

In the context of formal languages, a grammar is defined ([36], Chapter 4) as $G = (V, T, P, S)$, where

- $V$ = finite set of variables
- $T$ = finite set of terminals
- $P$ = finite set of production rules
- $S$ = start symbol ($S \in V$)

In a context-free grammar (CFG), production rules are of the form $A \rightarrow \alpha$, where $A$ is a variable and $\alpha$ is a string of symbols from $(V \cup T)^*$ [36].

10.2 $T*V*$ Form

In order to more easily model and process grammars, this research required the right-hand side of the production rules to be of the form $T*V*$. That is, the right-hand side of each production rule must contain first any number of terminals followed by any number of variables; terminals are not allowed to follow variables in a grammar rule. When grammars are in $T*V*$ format, all terminals and variables are each grouped together, which makes it easier to model the grammars with SDDL pools. Any CFG can be transformed to $T*V*$ form through the following algorithm:

- Identify rules where terminals follow variables
• For each such rule, suppose that the offending terminal was $\alpha$. To conform to $T^*V^*$ form,
  
  o Make a new rule with a new variable ($\beta$, for example) mapping to $\alpha$.
  
  o Substitute variable $\beta$ for terminal $\alpha$ in the non-conforming rule.

Consider the following sample grammar, where upper-case letters represent variables and lower-case letters represent terminals:

\[
\begin{align*}
S & \rightarrow AbBb \\
A & \rightarrow ab \\
A & \rightarrow Aa \\
B & \rightarrow ba \\
B & \rightarrow Bb
\end{align*}
\]

The grammar above has three rules (lines 1, 3 and 5) where terminals follow variables on the right-hand side, and so it is not in $T^*V^*$ form. However, it can easily be converted to $T^*V^*$ form by the “new rule creation and variable substitution” previously described:

\[
\begin{align*}
S & \rightarrow AYBY \\
A & \rightarrow ab \\
A & \rightarrow AX \\
B & \rightarrow ba \\
B & \rightarrow BY \\
X & \rightarrow a \\
Y & \rightarrow b
\end{align*}
\]

New rules 6 and 7 were created, and variables $X$ and $Y$ were substituted for terminals $a$ and $b$ (respectively) where necessary (rules 1, 3 and 5) to preserve $T^*V^*$ form.
10.3 Representing CFGs as Pools

Before modeling a CFG as a pool, the grammar must first be converted into $T^*V^*$ format. After that, a pool can be constructed to model the converted grammar.

In order to be processed correctly, a pool representing a CFG must follow a specific format. The top-level pool is named $CFG$. There is one choice in the $CFG$ pool for each variable in the grammar, and the choices are named after their respective variables. The top-level pool choices also include a sub-pool named $rules$. The $rules$ sub-pool for variable $X$ contains information about all of the rules in the grammar having $X$ as their left-hand side. Each choice in the $rules$ sub-pool has a unique (within that sub-pool) rule number as a name, and contains $yield$ and $repl$ (for “replace”) auxiliary values. The $yield$ value corresponds to the $T^*$ portion (terminals) of the rule, and the $repl$ value corresponds to the $V^*$ portion (variables) of the rule.

Consider, for example, the language $L=\{0^n1^n, \ n \geq 1\}$. This language can be represented by the following grammar:

\[
S \rightarrow 0A1 \\
A \rightarrow 0A1 \mid \varepsilon
\]

The grammar above is not yet in $T^*V^*$ form, since terminals follow variables on the right-hand sides of some rules. It can be transformed to $T^*V^*$ form as follows:

\[
S \rightarrow 0AB \\
A \rightarrow 0AB \mid \varepsilon \\
B \rightarrow 1
\]

The transformed grammar can then be modeled as a pool as follows:

<pool name="CFG">
  <choice name="S">
    <pool name="rules">
      <!-- Pool contents here -->
    </pool>
  </choice>
</pool>
10.4 SDDL Generation Code for CFGs

The SDDL code used to convert a CFG-pool to valid grammar strings looks like this:

```xml
<table name="cfg_values">
  <variable name="threadno" type="int">
    <iteration base="0" count="5">
      <itervar name="symbol" type="string" init=""S""/>
      <itervar name="stack" type="string" init="""/>
      <itervar name="str" type="string" init="""/>
    </iteration>
  </variable>
  <variable name="slen" type="int">
```
The following table contains explanations for each field/variable in the `cfg_values` table definition:

<table>
<thead>
<tr>
<th>Field/Variable</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>threadno</td>
<td>Allows for 5 generation threads. To change the number of generation threads, alter the “count” value.</td>
</tr>
</tbody>
</table>

The iteration constraint contains the following iteration variables:
- `symbol`: the current left-hand-side variable, initialized to “S”, which by convention is always the start symbol
- `stack`: the current contents of the stack, initialized to <empty>
- `str`: the current output-string value, initialized to <empty>
Iterates from 1 to 300 for each generation thread. This puts a ceiling on the size of the legal string that can be generated for the grammar. If you want to change the ceiling from 300, then alter the count value here.

Chooses one of the rules having symbol on the left-hand side.

Updates str variable by adding the yield (or terminal portion) associated with the rule field to str, and outputs new str value. This field is typed as an int (instead of a string) to suppress the quotes surrounding the output value.

Updates the stack variable. It first pops the top symbol off the stack, then pushes the repl variables (or variable portion) associated with the rule field onto the stack.

Set to true if the stack variable is empty, which denotes an acceptance state for the current string; otherwise, set to false.

If valid is true, then reset the symbol value to S, the start symbol. Otherwise, set symbol to the symbol on top of the stack.

If valid is true, then reset the str value to <empty>; otherwise, suppress the output of the current row.

Table 10-1: Field Explanations for cfg_values Table Definition

Using our $L=\{0^n1^n, \ n\geq 1\}$ grammar and CFG-pool from the previous section, generation within a thread might proceed as follows (iteration variables in italics, output values in bold, str initialized to “”, symbol initialized to S, stack initialized to “”):

<table>
<thead>
<tr>
<th>thread no</th>
<th>sle n</th>
<th>rule</th>
<th>stringval</th>
<th>stackop</th>
<th>valid</th>
<th>bump symbol</th>
<th>resetstring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>str=’”0”</td>
<td>stack=AB</td>
<td>false</td>
<td>symbol=A</td>
<td>Suppress row</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>str=’”0”</td>
<td>stack=ABB</td>
<td>false</td>
<td>symbol=A</td>
<td>Suppress row</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>str=’”0”</td>
<td>stack=BB</td>
<td>false</td>
<td>symbol=B</td>
<td>Suppress row</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>str=’”001”</td>
<td>stack=B</td>
<td>false</td>
<td>symbol=B</td>
<td>Suppress row</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td>str=’”0011”</td>
<td>stack=</td>
<td>true</td>
<td>symbol=S</td>
<td>str=’”’</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
<td>str=’”0”</td>
<td>stack=AB</td>
<td>false</td>
<td>symbol=A</td>
<td>Suppress row</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
<td>str=’”0”</td>
<td>stack=B</td>
<td>false</td>
<td>symbol=B</td>
<td>Suppress row</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
<td>str=’”01”</td>
<td>stack=</td>
<td>true</td>
<td>symbol=S</td>
<td>str=’”’</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>0</td>
<td>str=’”0”</td>
<td>stack=AB</td>
<td>false</td>
<td>symbol=A</td>
<td>Suppress row</td>
</tr>
</tbody>
</table>

Table 10-2: Sample Generation Sequence for $L=\{0^n1^n, \ n\geq 1\}$
When the CFG string generator recognizes and outputs a valid string, it resets the
\textit{str} value to \texttt{<empty>} and the \textit{symbol} value to the start symbol. In essence, the CFG
string generator restarts the derivation process when it outputs a valid string.

10.5 Example: Strings with an Even Number of 0s and 1s

This language is actually a regular language, and could be represented by the
deterministic finite automaton (DFA) shown below:

![DFA Diagram]

Figure 10-1: "Even Number of Zeroes and Even Number of Ones" DFA

The DFA above could be translated into a CFG as follows (where variables $S \rightarrow V$
correspond to states $Q0 \rightarrow Q3$):

\[
\begin{align*}
S \rightarrow & \quad \epsilon \\
S \rightarrow & \quad 0T \\
S \rightarrow & \quad 1U \\
T \rightarrow & \quad 0S \\
T \rightarrow & \quad 1V
\end{align*}
\]
Since the grammar above is already in $T^*V^*$ format, it can be represented in pool form as follows:

```xml
<pool name="CFG">
   <choice name="S">
      <pool name="rules">
         <choice name="1">
            <yield></yield>
            <repl></repl>
         </choice>
         <choice name="2">
            <yield>0</yield>
            <repl>T</repl>
         </choice>
         <choice name="3">
            <yield>1</yield>
            <repl>U</repl>
         </choice>
      </pool>
   </choice>
   <choice name="T">
      <pool name="rules">
         <choice name="1">
            <yield>0</yield>
            <repl>S</repl>
         </choice>
         <choice name="2">
            <yield>1</yield>
            <repl>V</repl>
         </choice>
      </pool>
   </choice>
   <choice name="U">
      <pool name="rules">
         <choice name="1">
            <yield>0</yield>
            <repl>V</repl>
         </choice>
         <choice name="2">
            <yield>1</yield>
            <repl>S</repl>
         </choice>
      </pool>
   </choice>
</pool>
```
The first 20 strings produced from this grammar were as follows:

<empty>
11011011
11
<empty>
00
<empty>
1001011100010110011110
110100000111101100100101110001000111001001011100010001110001010
1100
00100010
10.6 Example: Palindromes

The following grammar accepts palindromes over \( \{0, 1\}^* \):

\[
S \rightarrow \varepsilon
\]
The grammar above can easily be converted to $T^*V^*$ format:

\[
\begin{align*}
S & \rightarrow 0 \\
S & \rightarrow 1 \\
S & \rightarrow 0S0 \\
S & \rightarrow 1S1 \\
\end{align*}
\]

The grammar is represented in pool form as follows:

```xml
<pool name="CFG">
  <choice name="S">
    <pool name="rules">
      <choice name="1">
        <yield></yield>
        <repl></repl>
      </choice>
      <choice name="2">
        <yield>0</yield>
        <repl></repl>
      </choice>
      <choice name="3">
        <yield>1</yield>
        <repl></repl>
      </choice>
      <choice name="4">
        <yield>0</yield>
        <repl>SA</repl>
      </choice>
      <choice name="5">
        <yield>1</yield>
        <repl>SB</repl>
      </choice>
    </pool>
  </choice>
  <choice name="A">
    <pool name="rules">
      <choice name="1">
        <yield>0</yield>
      </choice>
    </pool>
  </choice>
  <choice name="B">
    <pool name="rules">
      <choice name="1">
        <yield>0</yield>
      </choice>
    </pool>
  </choice>
</pool>
```
When we run the pool above through our CFG string generation code, the first 20 valid strings produced look like this:

```
1
00000
1
1
1
111
0
<empty>
00
<empty>
0
101
0
1
101
11011
1
000
1
11
```

While these are all valid palindromes, they are all very short and uninteresting. In fact, of the 652 palindromes produced, only 4 were 10 characters long or longer. The reason for this is that all of the $S$ rules have an equal chance of being selected during
string construction. What if we assigned a higher weight to “complex” S-rules than to the “simple” S-rules? Let’s change our “complex” S-rules from this:

```xml
<choice name="4">
  <yield>0</yield>
  <repl>SA</repl>
</choice>
<choice name="5">
  <yield>1</yield>
  <repl>SB</repl>
</choice>
```

to this (changes in **bold**):

```xml
<choice name="4">
  <weight>9</weight>
  <yield>0</yield>
  <repl>SA</repl>
</choice>
<choice name="5">
  <weight>9</weight>
  <yield>1</yield>
  <repl>SB</repl>
</choice>
```

Since choice weights default to 1, there is now an 18-in-21 chance that a “complex” S-rule will be chosen, as opposed to a “simple” S-rule. Using this enhanced grammar, the first 20 valid strings generated now look like this:

```
011011010100010101110110
1100000011
001010100
11
01111100100110001110001110001100100100100111110
000001110000
1100010001111000100011
101101
0000
101
01010001010
0101010
10100101
```

239
The enhanced pool yielded only 113 palindromes, as opposed to the 652 palindromes generated by the original pool. However, the palindromes from the altered CFG pool are generally longer and more interesting than their counterparts from the original pool. In this fashion, the characteristics of generated strings can be manipulated by weighting rules in the grammar.

10.7 Example: Mathematical Expressions

The following grammar will generate mathematical expressions:

\[
S \rightarrow E \\
E \rightarrow EDE \mid (E) \mid P \\
D \rightarrow + \mid * \\
P \rightarrow N \mid V \\
N \rightarrow 1 \mid 2 \mid 3 \mid 4 \\
V \rightarrow a \mid b \mid c \mid d
\]

The operations supported in the grammar above include *, +, and (). Primaries can be variables (a-d) or numbers (1-4). The grammar is nearly in \(T^*V^*\) form and only needs a small modification (to the E-rules) to conform:

\[
S \rightarrow E \\
E \rightarrow EDE \mid (EA) \mid P \\
D \rightarrow + \mid * \\
P \rightarrow N \mid V \\
N \rightarrow 1 \mid 2 \mid 3 \mid 4 \\
V \rightarrow a \mid b \mid c \mid d \\
A \rightarrow )
\]
The converted grammar above can be modeled as a pool as follows:

```xml
<pool name="CFG">
  <choice name="S">
    <pool name="rules">
      <choice name="1">
        <yield/>
        <repl>E</repl>
      </choice>
    </pool>
  </choice>
  <choice name="E">
    <pool name="rules">
      <choice name="1">
        <yield/>
        <repl>EDE</repl>
      </choice>
      <choice name="2">
        <yield>(</yield>
        <repl>EA</repl>
      </choice>
      <choice name="3">
        <yield/>
        <repl>P</repl>
      </choice>
    </pool>
  </choice>
  <choice name="D">
    <pool name="rules">
      <choice name="1">
        <yield>+</yield>
        <repl></repl>
      </choice>
      <choice name="2">
        <yield>*</yield>
        <repl></repl>
      </choice>
    </pool>
  </choice>
  <choice name="P">
    <pool name="rules">
      <choice name="1">
        <yield></yield>
        <repl>N</repl>
      </choice>
      <choice name="2">
        <yield></yield>
      </choice>
    </pool>
  </choice>
</pool>
```
When run through the CFG string generator, the pool above only produced 10 legal expressions:

\[
\begin{align*}
\text{c} &\quad (((c+2*(1*(2)+b)*4)*((a)+((2)*(a+b)*((d)+1*(((b)*b+4)))*(c)*(d)))) \\
&\quad ((((((((4))*4))*((d)*3)*3))+(c)*((c+4)*3+4+(a))*3* \\
&\quad (((a))*(3))+(a)))*(2*a*3+d)*1 \\
&\quad 2 \\
&\quad 2 \\
&\quad ((((a)))+((((b+1*4+d)+3*(3))*a))*4)+1+(3)+c+a*b \\
\text{b} &\quad 4 \\
&\quad (1) \\
&\quad (3*1)
\end{align*}
\]

There are several instances of one or more sets of parentheses being placed around a single primary in the expressions generated above. The parenthesis rule in the original grammar could be changed from \( E \rightarrow (E) \) to \( E \rightarrow (EDE) \). In other words, make sure that parentheses surround an actual operation, instead of just a primary. In the transformed grammar, we would need to change \( E \rightarrow (EA \) to \( E \rightarrow (EDEA \). In our pool, we would change choice 2 of the \( E \)-rules from (change in **BOLD**):

\[
<\text{choice name="2"}> \\
<\text{yield}> </\text{yield}> \\
<\text{repl}>EA </\text{repl}> \\
</\text{choice}>
\]

to:

\[
<\text{choice name="2"}> \\
<\text{yield}> </\text{yield}> \\
<\text{repl}>EDEA </\text{repl}> \\
</\text{choice}>
\]
After making the change above, the generated legal expressions now look like:

c
2
2

Disappointing! Apparently, the change made the expression construction so complicated that anything besides a simple primary took longer than 300 characters (imposed as a limit by the \textit{slen} variable of the CFG SDDL generation code) to resolve itself. What if another change was made, to make it easier for expressions to become resolved within 300 characters? The \textit{E} \rightarrow \textit{P} rule could be weighted to be chosen more often, which would statistically cause expressions to resolve themselves sooner. We would need to change choice 3 of the \textit{E}-rules from

\begin{verbatim}
<choice name="3">
  <yield/>
  <repl>P</repl>
</choice>
\end{verbatim}

to

\begin{verbatim}
<choice name="3">
  <weight>2</weight>
  <yield/>
  <repl>P</repl>
</choice>
\end{verbatim}

Having made the above change, the generator now produced 35 legal expressions, the first 10 of which were:

d
(d*2)
(d* ((c*1)*2) + (b*4))
a+4
b
d
1+2+(1+2)*(3+a)
l
((c*(d+c)*(d+d)+2)*c)
This is a nice mix of simple and complex expressions, with no more annoying instances of singleton primaries being surrounded by parentheses. Again, the characteristics of the generated strings can be controlled by altering weights and making subtle changes to the grammar.

10.8 An Alternative Method for Generating Strings

The method used to parse grammars and generate legal strings that has been used to this point was described in Section 10.4. One could also use a plug-in function for grammar parsing.

Consider the following plug-in function definition:

```java
class legalstring implements PluginFunctionInterface {
    public Lexeme call(ArrayList<Lexeme> params, Environment env, RngInterface rnd)
            throws EvaluationException {
        // Check for presence of 2 parameters.
        if (params.size() != 2)
            throw new EvaluationException("legalstring(): Expected 2 parameters.");

        // Evaluate all parameters
        Lexeme ePoolname = params.get(0).evaluate(env);
        Lexeme eStartsym = params.get(1).evaluate(env);

        // Check validity of minLen, maxLen, minChar, maxChar
        if (ePoolname.getType() != Lexeme.STRING)
            throw new EvaluationException("legalstring(): Arg 1 (poolname) needs to be a STRING");

        String poolname = ePoolname.getStringValue();
    }
}
```
if (eStartsym.getType() != Lexeme.STRING) {
    throw new EvaluationException(
        "legalstring(): Arg 2 (startsym) needs to be a STRING");
}
String startsym = eStartsym.getStringValue();

// Find the pool
sdg.Pool p = (sdg.Pool) env.findElement(poolname);
if (p == null)
    throw new EvaluationException(
        "legalstring(): Failed to look up pool " + poolname);

// Initialize variables
String str = "";
String stack = "";
PoolChoice choice = p.getChoice(startsym);
Pool rules = choice.getSubPool("rules");
String rulename = rules.generateValue(rnd);
PoolChoice rulechoice = rules.getChoice(rulename);
stack = rulechoice.getAuxData("repl")+stack;
str = str + rulechoice.getAuxData("yield");

// Iterate through grammar rules while stack is non-empty
// while (stack.length() > 0)
{
    String newvar = stack.substring(0,1);
    stack = stack.substring(1);
    choice = p.getChoice(newvar);
    rules = choice.getSubPool("rules");
    rulename = rules.generateValue(rnd);
    rulechoice = rules.getChoice(rulename);
    stack = rulechoice.getAuxData("repl")+stack;
    str = str + rulechoice.getAuxData("yield");
}

// Empty stack means we now have a legal string (str).

// Construct a lexeme, fill it with our value, and return.
Lexeme rval = new Lexeme();
rval.setValue(str);
rval.setType(Lexeme.STRING);
return rval;
}

Using the plug-in function above, the table definition to generate legal strings for a grammar can be reduced to:

<database>
    <load funcName="legalstrings"/>
    [... CFG definition or reference ...]
    <table name="cfg_strings" length="100">

The advantages of such a plug-in function solution for generating legal strings for user-defined CFGs are that the SDDL table definition is much simpler, and also that one can specify the exact number of strings to generate, as each call to legalstring() results in exactly one legal string. The disadvantage of such a solution is that it takes a good deal of SDG architectural knowledge to code up such a function. Also, there are no safeguards against “runaway” grammars (i.e., grammars with left-recursion) as there were in the SDDL solution – some grammars might result in the legalstring() function running forever, thus hanging the SDG application. However, the “runaway” problem could be resolved by adding more logic to the plug-in function.

10.9 Summary

In this chapter, techniques have been demonstrated for generating legal strings for arbitrary context-free grammars (CFGs) using a general-purpose synthetic data generator (SDG). A standard representation format for CFGs in the form of SDDL pools was established. Two methods were presented for converting grammar pools to legal strings: the first was an approach that relied heavily on SDDL functionality, and the second approach used plug-in functions. While the examples given herein have been relatively modest, this proven ability to generate complex text according to user-defined constraints serves as a launching pad for more ambitious textual generation applications.
11 APPLICATION: GENERATING A MAILING LIST

A problem fairly common among retailers is the maintenance of customer mailing lists. Such mailing lists are typically large, and may contain a number of redundant entries for individuals. Periodically, a mailing list will need to be “scrubbed”, so that redundancies are weeded out and entry formats are homogenized. In order to test such “scrubbing” algorithms, it would be helpful if one could synthetically generate a mailing list with realistic properties (and shortcomings).

This chapter describes how SDG can be used to generate realistic mailing lists. This problem is approached as a two-phase process. First, one needs to generate a synthetic (but realistic) population. Next, one can generate a mailing list based upon the synthetically generated population.

To gain a full understanding of the SDDL table definitions used to produce these data sets, we will first explore some of the auxiliary pools used in their construction.

11.1 Auxiliary Pool Files

This section describes some of the auxiliary pools used during the generation of various data sets in this chapter.

11.1.1 StateZip.xml

The StateZip.xml file contains the StateZip pool, which is a multi-level pool derived from information gathered during the 2000 Census. This pool is used to synthesize accurate city, state and zip code information. At the top level are U.S. states and territories. Within each state choice is a zip code pool. Each zip code choice
contains city, county and population information about that zip code. Each zip code choice is weighted by population, and each state choice is weighted by population. Such weighting insures that the distribution of state and zip choice synthesized from the pool will accurately model the U.S. population distribution.

This pool is quite large (7+ MB) of text, but this abbreviated version shows the general structure of the StateZip pool:

```xml
<pool name="StateZip">
  <choice name="AK">
    <pool name="zips">
      <choice name="99501">
        <city>ANCHORAGE</city>
        <county>ANCHORAGE</county>
        <pop>16211</pop>
        <weight>16211</weight>
      </choice>
      <choice name="99502">
        <city>ANCHORAGE</city>
        <county>ANCHORAGE</county>
        <pop>18626</pop>
        <weight>18626</weight>
      </choice>
      [...]
    </pool>
    <pop>624992</pop>
    <weight>624992</weight>
  </choice> <!--end of "AK"/>
  <choice name="AL">
    <pool name="zips">
      [...]
    </pool>
    <pop>624992</pop>
    <weight>624992</weight>
  </choice> <!--end of "AL"/>
</pool>
```
Note that each `zips` choice has identical `pop` and `weight` auxiliary values, as do each of the top-level choices. The `weight` item is actually a keyword to weight the pool choice, and the `pop` item allows the user to synthesize population information. The `weight` item is not visible to the user.

11.1.2 names.xml

The names.xml file contains the `names` pool, which contains common first and last names from a number of ethnic groups. The ethnic groups supported are Indian, Japanese, Caucasian, Chinese, and Hispanic. Each of those choices has a `lastnames` pool and a `firstnames` pool, which contain popular last and first names (respectively) from the given ethnicity. In addition, for each entry in the Caucasian `firstnames` pool, there is a `variations` sub-pool with possible variations on the first name.

The following is an abbreviated representation of the `names` pool:

```
<pool name="names">
  <!--Indian last and first names-->
  <choice name="Indian">
    <!--www.answers.com/topic/list-of-most-common-surnames-->
    <pool name="lastnames">
      <choice name="Aggarwal"/>
      <choice name="Amin"/>
      <choice name="Banerjee"/>
      <choice name="Bhati"/>
      [...
    </pool>
  <!--http://www.20000-names.com/female_hindi_names.htm-->
  </choice>
  <!--http://www.20000-names.com/male_hindi_names.htm-->
  <pool name="firstnames">
    <choice name="Akhila"/>
    <choice name="Amulya"/>
    <choice name="Anisha"/>
    [...
  </pool>
</pool>
```
11.1.3 streetnames.xml

The streetnames.xml file is composed of three separate pools: the streetnames pool, the streettypes pool, and the directions pool.

11.1.3.1 streetnames pool

The streetnames pool contains as its choices a number of possible street names: U.S. presidents, generals, common plant and animal names, and generic names like state, university, main, etc... While the streetnames pool could be more rigorously derived by selecting street names from actual street name data sets, this
method of creating the streetnames pool was judged to produce sufficiently realistic street names.

11.1.3.2 streettypes pool

The streettypes pool contains common street types, along with variations on those types, and is defined as follows:

```xml
<pool name="streettypes">
  <choice name="Street">
    <pool name="variations">
      <choice name="Street"/>
      <choice name="St."/>
    </pool>
  </choice>
  <choice name="Avenue">
    <pool name="variations">
      <choice name="Avenue"/>
      <choice name="Ave."/>
    </pool>
  </choice>
  <choice name="Boulevard">
    <pool name="variations">
      <choice name="Boulevard"/>
      <choice name="Blvd."/>
    </pool>
  </choice>
  <choice name="Expressway">
    <pool name="variations">
      <choice name="Expressway"/>
      <choice name="Expwy."/>
    </pool>
  </choice>
  <choice name="Road">
    <pool name="variations">
      <choice name="Road"/>
      <choice name="Rd."/>
    </pool>
  </choice>
  <choice name="Court">
    <pool name="variations">
      <choice name="Court"/>
      <choice name="Ct."/>
    </pool>
  </choice>
</pool>
```
11.1.3.3 directions pool

The directions pool contains direction names and variations on those names:

```xml
<pool name="directions">
  <choice name="North">
    <pool name="variations">
      <choice name="North"/>
      <choice name="N."/>
    </pool>
  </choice>
  <choice name="South">
    <pool name="variations">
      <choice name="South"/>
      <choice name="S."/>
    </pool>
  </choice>
  <choice name="East">
    <pool name="variations">
      <choice name="East"/>
      <choice name="E."/>
    </pool>
  </choice>
  <choice name="West">
    <pool name="variations">
      <choice name="West"/>
      <choice name="W."/>
    </pool>
  </choice>
</pool>
```

11.1.4 StateAbbrToName.xml

The StateAbbrToName.xml file contains the StateAbbrToName pool. This pool allows for the extraction of a state's full name or old-style abbreviation, given the state's two-letter postal code. The pool looks like this:

```xml
<pool name="StateAbbrToName">
  <choice name="AK">
    <full>Alaska</full>
    <old>Alaska</old>
  </choice>
</pool>
```
11.2 Generating a Realistic Population

In order to be useful for the next step of producing a mailing list, the synthetic population needed to be produced in the form of an SDDL pool. Thus, the output from the generator needed to be in SDDL format. This section describes the SDDL table definition that was used to generate a realistic population in the form of an SDDL pool.

Note the use of XML escape sequences in the definitions below. It is illegal to use quotes, “greater than” or “less than” characters in an XML element or attribute value. To generate these characters, escape sequences must be used. Also, there are escape...
sequences available for various control characters. Some of these escape sequences are described in Table 11-1.

<table>
<thead>
<tr>
<th>Character</th>
<th>XML Escape Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>double-quote (&quot; )</td>
<td>&quot;</td>
</tr>
<tr>
<td>greater-than (&gt; )</td>
<td>&gt;</td>
</tr>
<tr>
<td>less-than (&lt; )</td>
<td>&lt;</td>
</tr>
<tr>
<td>carriage return</td>
<td></td>
</tr>
<tr>
<td>line feed</td>
<td>
</td>
</tr>
</tbody>
</table>

Table 11-1: XML Escape Sequences

Before the table definition comes the <database> start tag and some necessary preamble:

```xml
<database>
  <seed>5729947823</seed>
  <import filename="names.xml"/>
  <import filename="streetnames.xml"/>
  <import filename="StateZip.xml"/>

  <pool name="MiddleInitial">
    <choice name="A"/>
    <choice name="B"/>
    [...]
    <choice name="X"/>
    <choice name="Y"/>
    <choice name="Z"/>
    <choice name=" "/>
  </pool>

  <constant name="CRLF" type="string"
    value="&quot;&amp;#0013;&amp;#0010;&quot;"/>
  <constant name="HOUSEHOLDS" type="int"
    value="20"/>
</database>
```

The seed element allows for the seeding of the random number generator. The import elements import various files (pools) described in Section 11.1. The
MiddleInitial pool provides a means for synthesizing middle initials; note that the last choice is a blank, which signifies “no middle initial”. The “CRLF” string constant stores a carriage-return-line-feed XML sequence. Storing this sequence as a constant makes the ensuing table definition easier to read. The “HOUSEHOLDS” integer constant records the number of households that will be generated; for the purposes of this chapter, a relatively low number (20) is specified.

The table definition leads off with the table start element:

```xml
<table name="master" fileSuffix=".xml">
```

The outer iteration is contained in a variable called household:

```xml
<variable name="household" type="int">
<iteration base="1" count="HOUSEHOLDS">
<repeatMin>1</repeatMin>
<repeatMax>4</repeatMax>
<itervar name="refLastName" type="string" init="names[&quot;Caucasian&quot;].lastnames"/>
<itervar name="refStreetNum" type="int" init="1000+IRND(9000)"/>
<itervar name="refStreetDir" type="string" init="directions"/>
<itervar name="refStreetName" type="string" init="streetnames"/>
<itervar name="refStreetType" type="string" init="streettypes"/>
<itervar name="refState" type="string" init="&quot;AR&quot;"/>
<itervar name="refZip" type="string" init="StateZip[refState].zips"/>
<itervar name="refCity" type="string" init="StateZip[refState].zips[refZip].city"/>
<itervar name="related" type="bool" init="IRND(4)&gt;0"/>
<itervar name="index" type="int" init="0"/>
</iteration>
</variable>
```
Note the following about the iteration element above:

- The iteration continues for a count of “HOUSEHOLDS”.
- Each iteration element repeats anywhere from 1 to 4 times, meaning that each household has 1 to 4 inhabitants.
- The refLastName iteration variable stores a reference last name for each household. Note that, for illustrative purposes, we are only using Caucasian first and last names.
- The refStreetNum, refStreetDir, refStreetName and refStreetType iteration variables contain a reference street address for each household.
- The refState, refZip and refCity iteration variables contain a reference location for the household. For simplicity’s sake (for this demonstration), the state is limited to AR for Arkansas.
- The related iteration variable determines whether or not the household inhabitants are related to each other. As specified, there is a 3-in-4 chance that the household inhabitants are related.
- The index suffix keeps track of the number of household inhabitants.

The lastToUse variable holds the last name to use for the inhabitant being generated. If the household inhabitants are related, then this will be set to the contents of refLastName, or else a random last name will be chosen.

```xml
<variable name="lastToUse" type="string">
  <formula>
    related?refLastName:names["Caucasian"].lastnames
  </formula>
</variable>
```
The **lastname** variable holds an SDDL-formatted auxiliary element that contains last name information. If the last name were **Smith**, then this field would hold `<lastname>Smith</lastname>`.

```xml
<variable name="lastname" type="CHAR(100)">
  <formula>
    "&lt;lastname&gt;"+lastToUse+"&lt;/lastname&gt;"
  </formula>
</variable>
```

The **firstname** variable holds an SDDL-formatted auxiliary element that contains first name information. The data store here will look like `<firstname>Mary</firstname>`.

```xml
<variable name="firstname" type="CHAR(100)">
  <formula>
    "&lt;firstname&gt;"+
      names["Caucasian"][.firstnames+
      "&lt;/firstname&gt;"
  </formula>
</variable>
```

The **MI** variable holds a string, formatted as an SDDL auxiliary pool element, representing the middle initial. It is defined as `"<MI>E</MI>"`.

```xml
<variable name="MI" type="CHAR(32)">
  <formula>
    "&lt;MI&gt;"+MiddleInitial+"&lt;/MI&gt;"
  </formula>
</variable>
```

The **streetNum**, **streetDir**, **streetName** and **streetType** variables hold strings, formatted as SDDL auxiliary pool elements, representing the reference street address for the household. They might store values like `<streetNum> 1297`
</streetNum>, <streetDir> North </streetDir>, <streetName> Washington </streetName>, and <streetType> Avenue </streetType>, respectively.

The city, state and zip variables hold similar strings representing location information for the household. Values held in these variables might look like <city> FAYETTEVILLE </city>, <state> AR </state>, and <zip> 72701 </zip>, respectively.
The only output field defined in the table is called output. While the formula that defines this field may look complicated, it simply concatenates all of the previously calculated variables into a single string and outputs that string.

```
<field name="output" type="int">
  <formula>
    "&lt;choice name="+
    QUOTE(""+(household*10+(index=index+1)))+
    "&gt;"+CRLF+
    lastname+CRLF+
    firstname+CRLF+
    MI+CRLF+
    streetNum+CRLF+
    streetDir+CRLF+
    streetName+CRLF+
    streetType+CRLF+
    city+CRLF+
    state+CRLF+
    zip+CRLF+
    " &lt;/choice&gt;")
  </formula>
</field>
```

Finally, the table definition is closed:

```
</table>
```

When this table definition is run through the generator, a textual SDDL file named `master.xml` is output. The file looks like this\textsuperscript{15}:

```
<pool name="master">
```

\textsuperscript{15} Actually, the `<pool>` start and end tags need to be added by hand. It would be relatively easy to auto-generate them in single-generation-process mode, but not in multi-generation-process mode.
The master pool so generated can then be used as a master population from which a mailing list can be generated. The size of the file (i.e., the size of the synthetic population) will be determined by the value chosen for HOUSEHOLDS.

11.3 Generating a Master Population List

To generate a human-readable master population list from the previously generated master pool is relatively simple. The following SDDL code will accomplish the task:

<database>
  <import filename="master.xml"/>
</database>
<table name="masterlist" fileSuffix=".txt">
  <variable name="entry" type="string">
    <iteration pool="master"/>
  </variable>
  <variable name="fullname" type="CHAR(48)">
    <formula>
      ""+master[entry].lastname+",
      ""+master[entry].firstname+
      ""+master[entry].MI
    </formula>
  </variable>
  <variable name="streetaddr" type="CHAR(48)">
    <formula>
      ""+master[entry].streetNum+
      ""+master[entry].streetDir+
      ""+master[entry].streetName+
      ""+master[entry].streetType
    </formula>
  </variable>
  <variable name="location" type="CHAR(48)">
    <formula>
      ""+master[entry].city+",
      ""+master[entry].state+
      ""+master[entry].zip
    </formula>
  </variable>
  <field name="line" type="int">
    <formula>
      FILL(fullname,24,1," ")+
      FILL(streetaddr,38,1," ")+
      FILL(location,48,1," ")
    </formula>
  </field>
</table>
</database>

A file called masterlist.txt is generated. The definition iterates through all elements of the imported master.xml pool. The fullname variable combines the lastname, firstname and MI attributes of each element. The streetaddr variable combines the streetNum, streetDir, streetName and streetType attributes of each element. The location variable combines the city, state and
zip attributes of each element. The only output field is line, which concatenates the previously defined variables in a fixed-width format.

For our simple example, the generated master list looks like this:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>City, Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morison, Isabella K</td>
<td>6865 North Sixth Street</td>
<td>BAXITE, 72011</td>
</tr>
<tr>
<td>Wilson, Kayla W</td>
<td>9722 South Taft Expressway</td>
<td>WYNNE, 72396</td>
</tr>
<tr>
<td>Miller, Elizabeth U</td>
<td>1503 South Coolidge Court</td>
<td>SPRINGDALE, 72764</td>
</tr>
<tr>
<td>Wilson, Alexander A</td>
<td>1503 South Coolidge Court</td>
<td>SPRINGDALE, 72764</td>
</tr>
<tr>
<td>Campbell, Anna M</td>
<td>5700 West Harding Street</td>
<td>VILONIA, 72173</td>
</tr>
<tr>
<td>Watson, Susan N</td>
<td>5700 West Harding Street</td>
<td>VILONIA, 72173</td>
</tr>
<tr>
<td>Morison, Olivia Y</td>
<td>5700 West Harding Street</td>
<td>VILONIA, 72173</td>
</tr>
<tr>
<td>Thomson, Robert P</td>
<td>5700 West Harding Street</td>
<td>VILONIA, 72173</td>
</tr>
<tr>
<td>Jackson, Ryan E</td>
<td>3329 North Johnson Avenue</td>
<td>NORTH LITTLE ROCK, 72116</td>
</tr>
<tr>
<td>Morison, David N</td>
<td>2019 South Harrison Court</td>
<td>LITTLE ROCK, 72227</td>
</tr>
<tr>
<td>Morison, John U</td>
<td>2019 South Harrison Court</td>
<td>LITTLE ROCK, 72227</td>
</tr>
<tr>
<td>Morison, Brianna B</td>
<td>2019 South Harrison Court</td>
<td>LITTLE ROCK, 72227</td>
</tr>
<tr>
<td>Morison, Jessica V</td>
<td>2019 South Harrison Court</td>
<td>LITTLE ROCK, 72227</td>
</tr>
<tr>
<td>Thomas, Ethan X</td>
<td>1247 South Grant Street</td>
<td>BLYTHEVILLE, 72315</td>
</tr>
<tr>
<td>Thomas, Alexa I</td>
<td>1247 South Grant Street</td>
<td>BLYTHEVILLE, 72315</td>
</tr>
<tr>
<td>Thomas, Barbara V</td>
<td>1247 South Grant Street</td>
<td>BLYTHEVILLE, 72315</td>
</tr>
<tr>
<td>Thomas, Christopher J</td>
<td>1247 South Grant Street</td>
<td>BLYTHEVILLE, 72315</td>
</tr>
<tr>
<td>Moore, Abigail U</td>
<td>2796 North Johnson Avenue</td>
<td>ASHDOWN, 71822</td>
</tr>
<tr>
<td>Moore, Thomas C</td>
<td>2796 North Johnson Avenue</td>
<td>ASHDOWN, 71822</td>
</tr>
<tr>
<td>Clark, Elizabeth G</td>
<td>4337 North Fifth Avenue</td>
<td>DOVER, 72837</td>
</tr>
<tr>
<td>Macdonald, Alyssa S</td>
<td>7744 West Harrison Street</td>
<td>HOT SPRINGS NATIONAL PARK, 71901</td>
</tr>
<tr>
<td>Macdonald, John C</td>
<td>7744 West Harrison Street</td>
<td>HOT SPRINGS NATIONAL PARK, 71901</td>
</tr>
<tr>
<td>Macdonald, Jennifer S</td>
<td>7744 West Harrison Street</td>
<td>HOT SPRINGS NATIONAL PARK, 71901</td>
</tr>
<tr>
<td>Wilson, Madison N</td>
<td>2545 South Seventh Street</td>
<td>LITTLE ROCK, 72206</td>
</tr>
<tr>
<td>Wilson, Susan V</td>
<td>2545 South Seventh Street</td>
<td>LITTLE ROCK, 72206</td>
</tr>
<tr>
<td>Stewart, Alexander W</td>
<td>4996 West Center Street</td>
<td>HEBER SPRINGS VILLAGE, 72543</td>
</tr>
<tr>
<td>Stewart, Brandon B</td>
<td>4996 West Center Street</td>
<td>HEBER SPRINGS VILLAGE, 72543</td>
</tr>
<tr>
<td>Mitchell, William O</td>
<td>4748 East Eisenhower Court</td>
<td>MARSHALL, 72650</td>
</tr>
<tr>
<td>Johnson, Andrew X</td>
<td>4748 East Eisenhower Court</td>
<td>MARSHALL, 72650</td>
</tr>
<tr>
<td>Brown, Ashley V</td>
<td>4748 East Eisenhower Court</td>
<td>MARSHALL, 72650</td>
</tr>
<tr>
<td>Scott, Thomas S</td>
<td>4748 East Eisenhower Court</td>
<td>MARSHALL, 72650</td>
</tr>
<tr>
<td>Ross, Elizabeth H</td>
<td>2399 West Bowie Street</td>
<td>HEBER SPRINGS, 72543</td>
</tr>
<tr>
<td>Scott, Taylor R</td>
<td>2399 West Bowie Street</td>
<td>HEBER SPRINGS, 72543</td>
</tr>
<tr>
<td>Murray, Grace M</td>
<td>2399 West Bowie Street</td>
<td>HEBER SPRINGS, 72543</td>
</tr>
<tr>
<td>Campbell, Anthony U</td>
<td>6986 South Hoover Expressway</td>
<td>SPRINGDALE, 72764</td>
</tr>
<tr>
<td>Mitchell, Hannah</td>
<td>6986 South Hoover Expressway</td>
<td>SPRINGDALE, 72764</td>
</tr>
<tr>
<td>Robertson, William W</td>
<td>6986 South Hoover Expressway</td>
<td>SPRINGDALE, 72764</td>
</tr>
<tr>
<td>Stewart, Anthony I</td>
<td>6986 South Hoover Expressway</td>
<td>SPRINGDALE, 72764</td>
</tr>
<tr>
<td>Young, Elizabeth N</td>
<td>8941 West Ninth Boulevard</td>
<td>MAGNOLIA, 71753</td>
</tr>
<tr>
<td>Young, Joshua P</td>
<td>8941 West Ninth Boulevard</td>
<td>MAGNOLIA, 71753</td>
</tr>
<tr>
<td>Young, Steven T</td>
<td>8941 West Ninth Boulevard</td>
<td>MAGNOLIA, 71753</td>
</tr>
<tr>
<td>Young, Joshua H</td>
<td>8941 West Ninth Boulevard</td>
<td>MAGNOLIA, 71753</td>
</tr>
<tr>
<td>Anderson, Madison O</td>
<td>6947 South Ford Street</td>
<td>BENTONVILLE, 72712</td>
</tr>
<tr>
<td>Anderson, Dorothy R</td>
<td>6947 South Ford Street</td>
<td>BENTONVILLE, 72712</td>
</tr>
<tr>
<td>Anderson, Jessica</td>
<td>6947 South Ford Street</td>
<td>BENTONVILLE, 72712</td>
</tr>
<tr>
<td>Ross, Michael W</td>
<td>8496 North Ford Boulevard</td>
<td>MAUMELLE, 72113</td>
</tr>
<tr>
<td>Watson, Andrew G</td>
<td>2259 West Roosevelt Road</td>
<td>LITTLE ROCK, 72209</td>
</tr>
<tr>
<td>Brown, Richard O</td>
<td>6493 South Madison Expressway</td>
<td>HORSESHOE BEND, 72512</td>
</tr>
<tr>
<td>Brown, Richard G</td>
<td>6493 South Madison Expressway</td>
<td>HORSESHOE BEND, 72512</td>
</tr>
<tr>
<td>Brown, Elizabeth L</td>
<td>6493 South Madison Expressway</td>
<td>HORSESHOE BEND, 72512</td>
</tr>
<tr>
<td>Brown, Dorothy O</td>
<td>6493 South Madison Expressway</td>
<td>HORSESHOE BEND, 72512</td>
</tr>
</tbody>
</table>

A careful examination of the street addresses indicates that we do indeed have 20 households in our population, each household has anywhere from 1 to 4 inhabitants, and
most households are related (i.e., 6947 South Ford Street) but some are not (i.e., 2399 West Bowie Street).

11.4 Generating a Realistic Mailing List

A realistic customer prospects mailing list has a number of problems:

- The list may not contain fixed-width fields.
- The formats of entries in the list may vary. For example, some names may be listed with the last name first, and some names may be listed with the first name first.
- An individual may appear more than once in the list.
- An individual’s first name may appear in many different forms (i.e., William/Bill/Will).
- State names may be represented in a number of different fashions (i.e., “AR”, “Ark.”, “Arkansas”).
- Street directions and types may or may not be abbreviated.

A description of how to generate such a mailing list based on our synthetically generated population can now be provided.

First, the database tag and the imported pools (including the master list previously generated) appear:

```xml
<database>
  <import filename="master.xml"/>
  <import filename="names.xml"/>
  <import filename="StateAbbrToName.xml"/>
  <import filename="streetnames.xml"/>
</database>
```

Next comes the opening table element:
Next comes the outer iteration element, which in this case means that 250 names will be generated for the mailing list. Considering that about 50 names are in the reference (synthetic) population, that means that each individual in the reference population will be repeated in the mailing list an average of 5 times.

```xml
<variable name="cnt" type="int">
  <iteration base="1" count="250"/>
</variable>
```

The `entry` variable chooses a random entry from the reference population (the master pool):

```xml
<variable name="entry" type="string">
  <formula>master</formula>
</variable>
```

The next set of variables defines the format of the name generated from the mailing list:

```xml
<variable name="fname1" type="string">
  <formula>master[entry].firstname</formula>
</variable>
<variable name="fname" type="string">
  <formula>
    names["Caucasian"].firstnames[fname1].variations
  </formula>
</variable>
<variable name="MI1" type="string">
  <formula>
    master[entry].MI
  </formula>
</variable>
<variable name="MIexists" type="bool">
  <formula>
    (MI1>="A")&&(MI1<="Z")
  </formula>
</variable>
```
The `fname1` variable simply extracts the first name from the randomly chosen population entry; the `fname` element chooses a random variation of `fname1`. `MI1` extracts the middle initial from the population entry; `MIexists` detects whether or not a middle initial exists, and `MI` constructs the final middle initial string. The `fullname` variable is the fully constructed name; it has a 2-in-3 chance of being in (last, first MI) format, and a 1-in-3 chance of being in (first MI last) format.

The next set of variables defines the format of the street address:

```xml
<variable name="sdirl1" type="string">
  <formula>
    master[entry].streetDir
  </formula>
</variable>
<variable name="sdir" type="string">
  <formula>
    directions[sdirl1].variations
  </formula>
</variable>
<variable name="stypel" type="string">
  <formula>
    master[entry].streetType
  </formula>
</variable>
<variable name="stype" type="string">
  <formula>
    streettypes[stypel].variations
  </formula>
</variable>
```
The \texttt{sdir1} and \texttt{stype1} variables extract the street direction and street type, respectively, from the master list entry. The \texttt{sdir} and \texttt{stype} variables hold variations on \texttt{sdir1} and \texttt{stype1}, respectively. The \texttt{streetaddr} variable holds a fully constructed street address incorporating the variations held in \texttt{sdir} and \texttt{stype}.

The next set of variables defines location format:

\begin{verbatim}
<variable name="state1" type="string">
  <formula>master[entry].state</formula>
</variable>
<variable name="stateRepChooser" type="int">
  <formula>IRND(3)</formula>
</variable>
<variable name="state" type="string">
  <formula>
    stateRepChooser==0?state1:stateRepChooser==1?
      StateAbbrToName[state1].full:
      StateAbbrToName[state1].old
  </formula>
</variable>
<variable name="location" type="CHAR(48)">
  <formula>
    ""+master[entry].city+, ""+state+ ""+master[entry].zip
  </formula>
</variable>
\end{verbatim}

The \texttt{state1} variable extracts the state from the master list entry. The \texttt{stateRepChooser} variable is a random number between 0 and 2 (inclusive) for the purpose of choosing a representation format for the state. The \texttt{state} variable will be set equal to \texttt{state1} (2-letter state abbreviation) if \texttt{stateRepChooser} is 0, will be set
equal to the full name of the state if \texttt{stateRepChooser} is 1, and will be the old-style state abbreviation if \texttt{stateRepChooser} is 2. The \texttt{location} variable will be a fully formatted location string, incorporating the state representation found in the \texttt{state} variable.

Finally, the only output field, named \texttt{line}, is defined. The \texttt{line} field concatenates the \texttt{fullname}, \texttt{streetaddr} and \texttt{location} variables. Note that the \texttt{line} field is typed as an \texttt{int} even though it is string data; this is a trick employed to remove the surrounding quotes from an output string.

\begin{verbatim}
<field name="line" type="int">
  <formula>
    fullname+" "+streetaddr+" "+location
  </formula>
</field>
\end{verbatim}

Then the table and database elements are closed out.

\begin{verbatim}
</table>
</database>
\end{verbatim}

When the SDDL definition described above is provided as input to SDG, a file called “errorlist.txt” is generated. The first 20 lines of this 250-line file look like this:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson, Andrew X</td>
<td>4748 East Eisenhower Ct.</td>
<td>MARSHALL, AR 72650</td>
</tr>
<tr>
<td>Morison, Libby Y</td>
<td>5700 W. Harding St.</td>
<td>VILONIA, Ark. 72173</td>
</tr>
<tr>
<td>Jackson, Ryan</td>
<td>3329 N. Johnson Ave.</td>
<td>NORTH LITTLE ROCK, Ark. 72116</td>
</tr>
<tr>
<td>Ross, Mike W</td>
<td>8496 North Ford Boulevard</td>
<td>MAUMELLE, Arkansas 72113</td>
</tr>
<tr>
<td>Taylor R Scott</td>
<td>2399 W. Bowie Street</td>
<td>HEBER SPRINGS, Ark. 72543</td>
</tr>
<tr>
<td>Gracie M Murray</td>
<td>2399 West Bowie Street</td>
<td>HEBER SPRINGS, AR 72543</td>
</tr>
<tr>
<td>T R Scott</td>
<td>2399 W. Bowie Street</td>
<td>HEBER SPRINGS, Arkansas 72543</td>
</tr>
<tr>
<td>Alexander A Wilson</td>
<td>1503 South Coolidge Court</td>
<td>SPRINGDALE, Arkansas 72764</td>
</tr>
<tr>
<td>Watson, S</td>
<td>5700 West Harding Street</td>
<td>VILONIA, AR 72173</td>
</tr>
<tr>
<td>Wilson, K</td>
<td>9722 South Taft Expwy.</td>
<td>WYNNE, Ark. 72396</td>
</tr>
<tr>
<td>G M Murray</td>
<td>2399 West Bowie St.</td>
<td>HEBER SPRINGS, Ark. 72543</td>
</tr>
<tr>
<td>Wilson, Susan</td>
<td>2545 S. Seventh Street</td>
<td>LITTLE ROCK, Arkansas 72206</td>
</tr>
<tr>
<td>Campbell, A</td>
<td>5700 West Harding St.</td>
<td>VILONIA, AR 72173</td>
</tr>
<tr>
<td>Morison, Isabella K</td>
<td>6865 North Sixth Street</td>
<td>BAUXITE, Ark. 72011</td>
</tr>
<tr>
<td>Jackson, Ryan</td>
<td>3329 North Johnson Avenue</td>
<td>NORTH LITTLE ROCK, Ark. 72116</td>
</tr>
<tr>
<td>Brown, Dorothy</td>
<td>6493 South Madison Expressway</td>
<td>HORSESHOE BEND, Ark. 72512</td>
</tr>
<tr>
<td>Johnson, Andrew X</td>
<td>4748 East Eisenhower Ct.</td>
<td>MARSHALL, Arkansas 72650</td>
</tr>
<tr>
<td>Morison, D N</td>
<td>2019 South Harrison Court</td>
<td>LITTLE ROCK, AR 72227</td>
</tr>
<tr>
<td>Brianna Morison</td>
<td>2019 South Harrison Court</td>
<td>LITTLE ROCK, AR 72227</td>
</tr>
<tr>
<td>Moore, Abby</td>
<td>2796 N. Johnson Avenue</td>
<td>ASHDOWN, Arkansas 71822</td>
</tr>
</tbody>
</table>
Note that, even in these first few lines, some individuals are duplicated (Taylor Scott, Gracie Murray, Andrew Johnson). Some names are in (last, first MI) format, while some are in (first MI last) format. Some middle initials are omitted. First names can be in a number of formats (for example, Taylor/T or Gracie/G). Street directions and types may or may not be abbreviated. And the state (Arkansas) is represented in 3 different formats (Arkansas, AR, and Ark.). All of these properties make this a realistic mailing list.

11.5 Summary

This chapter described how SDG/SDDL can be used to synthetically generate a realistic mailing list. Though the data sets generated in this chapter were relatively small, they could easily be scaled up in size to be useful for real-world testing. Also, with a little more research, the synthetically generated population could contain first and last names (along with first-name variations) from additional ethnicities.

This application was significant for a few reasons:

- Mailing list processing is an important real-world problem. Some companies specialize in mapping raw mailing lists containing wide variations into canonical mailing lists. The generation of synthetic mailing lists can aid in the development of algorithms to automate the mapping from raw to canonical form by providing paired data files, one with raw and the other with canonical name and address lists for recognition algorithm regression testing.
• The intermediate file, containing a synthetic population, was essentially an XML file. Thus, it was demonstrated that SDG/SDDL can be used to produce data in an arbitrary well-formed language (in this case, XML).

• This mailing list application was another example where the output was textual as opposed to tabular, showing again that synthetic data generation can be used for applications outside of the purely tabular domain.
12 CONCLUSION

The research reported herein resulted in the development of the Synthetic Data Description Language (SDDL) as well as a synthetic data generation engine (SDG) that accepts SDDL files as input and is capable of generating data in parallel. In this chapter, conclusions are drawn based upon the development and performance of this synthetic data generation framework.

12.1 Veracity of Thesis Statement

The thesis statement given in chapter 1 of this dissertation can be broken down into four parts:

1. Synthetic data generation has theoretical underpinnings in the areas of random number theory, database representational concepts and data obfuscation.
2. Data generation constraints can be captured in a well-defined description language.
3. A synthetic data generator can be designed to execute efficiently and run in parallel.
4. Synthetic data generation has applications in the traditional tabular domain and beyond.

Part (1) was shown to be true in Chapters 5, 6 and 7. Chapter 5 demonstrated that SDDL was capable of enforcing common constraint types associated with the relational and entity-relationship data models. Chapter 6 described the care taken to make sure that generated random data was truly random. Chapter 7 outlined some techniques for using
SDDL to obfuscate data before exporting it to a third party for analysis or application development.

Parts (2) and (3) had already been proven true separately before this research was performed. General-purpose synthetic data generation frameworks had already been developed that could efficiently generate data in parallel (most notably, [15] and [9]). Also, at least one rich language [13] had already been developed for describing the characteristics of synthetic data. However, the two concepts had never before been combined into a single framework, as this research accomplished. The SDG/SDDL framework is simultaneously capable of both rich, flexible data description (SDDL, Chapter 2) and easy, efficient parallelization (Chapter 4). To the author’s knowledge, this combination makes the SDG/SDDL framework unique.

Part (4) was shown to be true in the application chapters (Chapters 8-11). Chapters 8 and 9 described the use of SDG for generating tabular data sets with typical constraints associated with them, such as inter-table dependencies and business rules. Chapters 10 and 11 demonstrated the use of SDG for generating non-tabular, textual data sets.

12.2 Real-World Impact of Synthetic Data Generation Technology

Chapters 8-11 showcased SDG in response to some real-world applications. Indeed, there were other efforts not described in this dissertation. In December 2006, SDG was used to produce synthetic data simulating a simple RFID supply chain data. That data was used by SensorConnect, an Australian company, to test the efficiency of their RFID “backbone” or repository, and this effort resulted in a conference paper published in the first IEEE International Conference on RFID [37]. In early 2007, SDG
was used to produce simulated “work-flow grid” data for Acxiom-sponsored research being performed at the University of Arkansas. Also in 2007, SDG was used to generate data for another Acxiom problem which involved autogenerating equivalence classes of names (in different formats) where the datasets were too large to fit within a machine for a problem involving name matching in distributed environments. In addition, there were other SDG applications developed for other projects. In all instances, the research projects were considerably accelerated by access to quickly generated application-specific synthetic data.

Other groups have also shown interest. A health insurance company explored our ability to produce “realistic but not real” health care data for purposes of regression testing and third-party analysis. A nationwide consulting company inquired about our ability to produce synthetic human resources data. A large database vendor inquired about our ability to generate large synthetic datasets for their customers to test their product before purchase. Both SensorConnect and the RFID Research Center here at the University of Arkansas have expressed interest in synthetic generation of complex RFID supply chain data (more realistic versions of the kind of data set described in Section 4.2). We have not to this point been able to collaborate with all these organizations on their projects, but our experience shows that there is considerable demand for synthetic data generation.

12.3 Future Directions

During the course of this research, a number of potential future improvements to (or applications of) the SDG/SDDL framework have been suggested:

273
Graphical user interface. Although a graphical user interface for SDG was developed (by another student), it does not cover all functionality of the current SDG system. Extensions to cover all of SDG would make the entire SDG system more usable by other end-users than the SDG developer. That graphical user interface also provided an extract and populate capability when connected to a relational database via an ODBC connection. That is, it could extract schema information including table and field names and types. If this capability could also be extended to extract other database metadata, including cardinality of tables, join selectivity, and security and integrity constraints, that information could be used to guide (some of) the specification of SDDL specifications. In addition, access to real data would provide a way to populate pools – but care would be needed (as described in Chapter 7) to insure that the data generated does not include real world instances.

Automatic Obfuscation Enforcement. Could the SDG graphical user interface provide obfuscation sliders for each field, where the user could specify an obfuscation level, and the generation engine would automatically obfuscate the field to the degree specified? This would save some work for the user.

Other non-tabular applications. Chapter 10 showed that context-free languages could be described in the form of SDDL pools, and showed how to generate legal strings for arbitrary context-free languages. Would it also be possible to capture some music theory rules in SDDL, and
produce random musical passages? Would it be possible to describe geography and cartography rules in SDDL and produce synthetic landscapes? These questions might have some interesting answers, and are worth exploring.

- Streaming, real-time data. Currently, SDG has no concept of time; it simply produces data as fast as it can. To be more useful in simulation environments, the concept of time could be incorporated into both SDDL and SDG. Instead of specifying something like “I want X rows of data generated”, allow for a specification more like “I want N rows generated every second” or “I want X to Y milliseconds delay between each row output”. Along with this, it would be nice if streaming output were supported.

- Non-parallelizable constraint types. A few new constraint types in SDDL have been rejected because they do not easily parallelize. For example, it might be nice to have a pool with a “balanced” property, which guaranteed that each contained pool choice would be chosen once before one was chosen twice. But implementing such a construct in parallel would require inter-process communication, which the SDG parallelization model prohibits. However, not all applications require parallel data generation. Such a constraint type could be supported if the user understood that it would mean that parallel data generation would be prohibited.

- Intermediate functional language. While SDDL supports user-defined plug-in functions, these functions must be structured in a precise way, and
sometimes writing them requires considerable knowledge of SDG architecture. It might be useful to support a functional description language in SDDL; this would let users write functions without having to understand about the underlying architecture.

- **Inverse Data Mining.** Data mining is used to discover new relationships in data including clusters and categories. Several toolsets exist to mine data in relational databases. These relationships may be present in a real dataset and if SDG is used to model that data, the result may not reflect these hidden relationships. When these relationships are known and when it is desirable for SDG to generate data that reflects these relationships, then SDG will need to be extended to reflect these relationships. Many data mining operators are processing intensive but it may be that the inverse operators that SDG would need to employ are computationally more tractable, even straightforward. For instance, if real data is found by some data mining operator to cluster, then an inverse operator might be implemented by the data pools concept along with normal or other distributions in a fairly direct manner.

- **Composable Constraints.** In the current framework, each field in a table is allowed a single type of constraint: either a min/max constraint OR a statistical distribution constraint OR a formula OR a query pool OR an iteration. What if SDG/SDDL allowed for multiple constraints to be imposed on a single field? Could multiple ranges be OR-ed or AND-ed together? Could a function call be composed with a range? Would all
such compositions make sense? Would constraint composition violate in any way the determinism or parallelism that has been maintained by the framework to this point?

- Provide support for additional constraint types, including (but not necessarily limited to):
  - Extended E-R (EER) Model Constraints which includes class-subclass modeling and mutually exclusive sets.
  - Fuzzy Logic Constraints. Examples of these might be “Many of the employees should be tall”, or “Few salaries are over $80,000”.
  - Data Mining Cluster Constraints. Data mining clusters can currently be described using formulas and pools, but it would be nice if there was a more direct way to specify them.
  - Logical Assertions. Allow the user to specify logical assertions about the data, and then parse and enforce those assertions. For example, in the Hallux Member_Instrument table, it might have been nice to specify something like “Musicians in jazz bands do not play the harmonica” or “Corporate customers only order albums”. Such assertions can now be specified through complicated pool descriptions and manipulations, but it would be nice if there was a simple way to state such assertions.Assertions might be enforced while the data is being generated, or they might be enforced as a post-generation step.
  - E-R Diagram Importation. It can be tedious to manually enter SDDL constraints from an E-R diagram. What if such diagrams could be
automatically converted to SDDL descriptions? One obstacle here would be the non-uniformity of various E-R dialects; another would be the multiple storage formats of E-R diagrams. But if such functionality could be added, it would save significant time for the user.
BIBLIOGRAPHY


280


[26] “Linear Congruential Generator”,


[28] P. L’Ecuyer, source code for Java RngStream class,


[30] L’Ecuyer's Combined Recursive Generator,


A SDDL DOCUMENT TYPE DEFINITION (DTD)

<!ELEMENT database (seed? fieldSep? import* load* constant* pool* table+)>  
<!ELEMENT seed (#CDATA)>  
<!ELEMENT fieldSep (#CDATA)>  
<!ELEMENT import EMPTY>  
<!ATTLIST import filename CDATA #REQUIRED>  
<!ELEMENT load EMPTY>  
<!ATTLIST load funcName CDATA #REQUIRED>  
<!ELEMENT constant EMPTY>  
<!ATTLIST constant name CDATA #REQUIRED>  
<!ATTLIST constant type CDATA #REQUIRED>  
<!ATTLIST constant value CDATA #REQUIRED>  
<!ELEMENT pool (choice+)>  
<!ATTLIST pool name CDATA #REQUIRED>  
<!ELEMENT choice (pool|ANY)*>  
<!ATTLIST choice name CDATA #REQUIRED>  
<!ELEMENT table ((field | variable)+ sql*)>  
<!ATTLIST table name CDATA #REQUIRED>  
<!ATTLIST table length CDATA>  
<!ATTLIST table fileSuffix CDATA>  
<!ELEMENT sql (#CDATA)>  
<!ELEMENT field (formula|(min max step?)|dist|queryPool|iteration)>  
<!ATTLIST field name CDATA #REQUIRED>  
<!ATTLIST field type CDATA #REQUIRED>  
<!ELEMENT min (#CDATA)>  
<!ELEMENT max (#CDATA)>  
<!ELEMENT step (#CDATA)>  
<!ELEMENT dist (tier+)>  
<!ELEMENT tier EMPTY>  
<!ATTLIST tier prob CDATA #REQUIRED>
<!ATTLIST tier min CDATA #REQUIRED>
<!ATTLIST tier max CDATA #REQUIRED>

<!ELEMENT queryPool EMPTY>
<!ATTLIST queryPool query CDATA #REQUIRED>

<!ELEMENT iteration (((repeatMin repeatMax) | repeatDist)? itervar*)>
<!ATTLIST iteration base CDATA>
<!ATTLIST iteration count CDATA>
<!ATTLIST iteration query CDATA>
<!ATTLIST iteration pool CDATA>

<!ELEMENT repeatMin #CDATA>
<!ELEMENT repeatMax #CDATA>
<!ELEMENT repeatDist tier+>

<!ELEMENT itervar EMPTY>
<!ATTLIST itervar name CDATA #REQUIRED>
<!ATTLIST itervar type CDATA #REQUIRED>
<!ATTLIST itervar init CDATA #REQUIRED>

<!ELEMENT formula #CDATA>
One of the most powerful means of data definition and constraint provided by SDDL is the formula. Not only can formulas be used by formula constraints:

```xml
<field name="ship_date" type="date">
  <formula>order_date+IRND(4)</formula>
</field>
```

but they can also be used to define min/max values, as in the “last_raise_date” field below:

```xml
<field name="hire_date" type="date">
  <min>#1990-01-01#</min>
  <max>#1999-12-31#</max>
</field>
<field name="last_raise_date" type="date">
  <min>hire_date+180</min>
  <max>TODAY()</max>
</field>
```

This appendix provides a detailed description of the grammar and semantics associated with formula constraints.

### B.1 Formula Expression Grammar

The grammar for a formula expression is defined formally as follows (adapted from [38]):

```
expr       : expr15
expr15     : expr14 |
            NAME ASSIGN expr15
expr14     : expr13 |
            expr13 QUESTION expr14 COLON expr14
expr13     : expr12 (OR expr12)*
expr12     : expr8 (AND expr8)*
expr8      : expr7 ((EQUALS | NOTEQUALS) expr7)*
```

286
expr7 : expr5 ((LT | LE | GT | GE) expr5)*
expr5 : expr4 ((PLUS | MINUS) expr4)*
expr4 : expr2 ((TIMES | DIVIDES | MODULUS) expr2)*
expr2 : expr1 |
        NOT expr1
expr1 : LPAREN expr RPAREN |
        primary (SUBFIELD STRING)?
primary : INTEGER | REAL | STRING | DATE |
          TIME | TIMESTAMP | BOOL | NAME |
       funcall | poolref
funcall : NAME LPAREN (expr (COMMA expr)*)? RPAREN
poolref : (NAME OBRACKET expr CBRACKET [shifter] |
          PERIOD)* end
end : NAME [OBRACKET expr CBRACKET shifter]
shifter : (PLUS | MINUS) expr

From this grammar, operator precedence and associativity are established as follows (from lowest precedence to highest precedence):

<table>
<thead>
<tr>
<th>operator(s)</th>
<th>associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment (=)</td>
<td>R-L</td>
</tr>
<tr>
<td>ternary if-then-else</td>
<td>L-R</td>
</tr>
<tr>
<td>(a?b:c)</td>
<td></td>
</tr>
<tr>
<td>logical OR (</td>
<td></td>
</tr>
<tr>
<td>logical AND (&amp;&amp;)</td>
<td>L-R</td>
</tr>
<tr>
<td>Equals (==, !=)</td>
<td>--</td>
</tr>
<tr>
<td>Relational (&gt;,&lt;,&gt;=,&lt;=)</td>
<td>--</td>
</tr>
<tr>
<td>Addition (+,-)</td>
<td>L-R</td>
</tr>
<tr>
<td>Multiplication (*,/,%</td>
<td>L-R</td>
</tr>
<tr>
<td>logical NOT (!)</td>
<td>--</td>
</tr>
<tr>
<td>Parentheses,</td>
<td></td>
</tr>
<tr>
<td>subfield($)</td>
<td>--</td>
</tr>
</tbody>
</table>

Table B-1: Formula Operator Precedence and Associativity

There are a number of possible primaries in this grammar, as described by Table B-2:

<table>
<thead>
<tr>
<th>Primary</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>[0-9]+</td>
<td>integer base-10 constant</td>
</tr>
<tr>
<td>REAL</td>
<td>[0-9]+.[0-9]+</td>
<td>decimal base-10 constant</td>
</tr>
<tr>
<td>STRING</td>
<td>&quot;&lt;anything&gt;&quot;</td>
<td>quoted text</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TIME</td>
<td>#HH:MM:SS#</td>
<td>time constant, book-ended by # characters; can be 24-hour time</td>
</tr>
<tr>
<td>DATE</td>
<td>#YYYY-MM-DD#</td>
<td>date constant, book-ended by # characters</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>#YYYY-MM-DD HH:MM:SS#</td>
<td>timestamp constant (date and time), book-ended by # characters</td>
</tr>
<tr>
<td>NAME</td>
<td></td>
<td>name of another field/variable, iteration variable, or constant</td>
</tr>
</tbody>
</table>
| funcall | (see grammar) | Currently supported built-in functions include:  
- ABS(int)  
- ABS(real)  
- TODAY() returns current date  
- NOW() returns current timestamp  
- IRND(int n) returns a random integer between 0 and (n-1), inclusive.  
- TORADIX(int num, int base) returns a string representation of decimal number “num” in base “base”  
- SUPPRESS_ROW() suppresses the generation of the current row  
- FILL(int/string orig, int size, int filltype, string fillchar) returns a string of size “size” with populated by the original string “orig” and the and appropriate number of occurrences of the specified fill character “fillchar”. The filltype parameter must be –1 (fill left), 0 (center) or 1 (fill right). The fillchar parameter must be a single-character string.  
- DATE(string) casts the string as a date and returns the result in a date lexeme.  
- BOOL(string) casts the string as a bool and returns the result in a bool lexeme.  
- MANGLE(string) either adds a random character to a string, or replaces a character with a random character. The result is returned as a string lexeme.  
- BINTOHEX(string) converts the input binary string (all ‘0’ and ‘1’ chars) to a hex string, and returns the value as a string lexeme.  
- SPLIT(string,sep,index) tokenizes the given string according to separator sep, and returns the token corresponding to the index. |
TRIM(string) trims the white space from the beginning and the end of the string.

SDDL also supports user-supplied plug-in functions.

<table>
<thead>
<tr>
<th>Poolref</th>
<th>(see grammar)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Can denote a (nested) pool or an auxiliary attribute within a pool</td>
</tr>
</tbody>
</table>

Table B-2: Formula Primaries

### B.2 Formula Expression Semantics

The grammar above defines a syntactically correct expression for use with a formula. However, there may be some expressions that are syntactically correct but semantically faulty. For example, it is syntactically correct to divide a string by a date, but this does not make sense semantically and it is not supported. These semantic rules are enforced in the data generator itself.

Table B-3 describes the operations supported by each combination of data types ("relational" denotes “equal (==), not equal (!=), less than (<), greater than (>), less than or equal (<=), greater than or equal (>=)”, “LHS” = “left-hand side”, “RHS” = “right-hand side”):

<table>
<thead>
<tr>
<th>LHS RHS=</th>
<th>int</th>
<th>real</th>
<th>string</th>
<th>Date</th>
<th>time</th>
<th>time stamp</th>
<th>bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>+,-,</td>
<td>+,-,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+,-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>+,</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+,-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>+,-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+,-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>+,</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+,-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time stamp</td>
<td>+,-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
There are rules regarding the result types for the operations above. Table B-4 outlines the result rules for the addition of various data types:

<table>
<thead>
<tr>
<th>LHS</th>
<th>+</th>
<th>RHS:</th>
<th>int</th>
<th>real</th>
<th>String</th>
<th>date</th>
<th>time</th>
<th>time stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>real</td>
<td>int</td>
<td>real</td>
<td>int (RHS converted to int)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>real</td>
<td>real</td>
<td>real</td>
<td>real</td>
<td>real (RHS converted to real)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>string converted to string</td>
<td>string converted to string</td>
<td>String</td>
<td>string converted to string</td>
<td>string converted to string</td>
<td>string converted to string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>date</td>
<td>String</td>
<td>date converted to string</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>time</td>
<td>String</td>
<td>time converted to string</td>
<td></td>
<td>time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time stamp</td>
<td>time converted as seconds</td>
<td>time converted as seconds</td>
<td>time converted to string</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B-4: Result Rules for Addition Operation

Table B-5 describes result rules for the subtraction operation with various types as operands:

<table>
<thead>
<tr>
<th>LHS</th>
<th>-</th>
<th>RHS:</th>
<th>int</th>
<th>real</th>
<th>string</th>
<th>date</th>
<th>time</th>
<th>time stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>real</td>
<td>int</td>
<td>real</td>
<td>int (RHS converted to int)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>real</td>
<td>real</td>
<td>real</td>
<td>real</td>
<td>real (RHS converted to real)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table B-5: Result Rules for Subtraction Operation

<table>
<thead>
<tr>
<th>LHS</th>
<th>RHS</th>
<th>int (difference in days)</th>
<th>int (difference in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date (RHS interpreted as days)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>time (RHS interpreted as seconds)</td>
<td></td>
<td>int (difference in seconds)</td>
</tr>
<tr>
<td>time stamp</td>
<td>time (RHS interpreted as seconds)</td>
<td></td>
<td>int (difference in seconds)</td>
</tr>
</tbody>
</table>

Table B-6 describes the result rules for multiplication with various operand type combinations:

<table>
<thead>
<tr>
<th>LHS * RHS:</th>
<th>int</th>
<th>real</th>
<th>string</th>
<th>date</th>
<th>time</th>
<th>time stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>real</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>real</td>
<td>real</td>
<td>real</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time stamp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B-6: Result Rules for Multiplication Operation

B.3 Examples

The following are a number of operations combining various types, along with their results:

\[ 2 + 1 \times 4 \rightarrow 6 \]
(2 + 1) * 4 ➞ 12
“14” * 2 ➞ “1414”
2 * “14” ➞ 28
#3:32:00# * 2 ➞ #7:04:00#
#2001-11-17# - #2001-11-10# ➞ 7
#2002-07-19 01:35:17# - #2002-07-19 01:34:19# ➞ 58
“date is: ” + #2004-01-17# ➞ “date is: 2004-01-17”
#1998-06-10# + 4 ➞ #1998-06-14#
#12:22:47# + 9 ➞ #12:22:56#
C DETAILS OF HALLUX PROBLEM AND SOLUTION

This appendix gives details associated with the Hallux database generation. For each table, a description is given of the schema, additional business rules, and the resulting SDDL description of that table. Tables are listed in the order that they were generated.

C.1 Instrument

The Instrument table contains IDs and names for all instruments played by Hallux musicians.

C.1.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instrument_ID</td>
<td>Uniquely Identifies an Instrument (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Instrument_Name</td>
<td>The name of an instrument</td>
<td>CHAR(50)</td>
</tr>
</tbody>
</table>

C.1.2 Rules

None.

C.1.3 SDDL Description

```xml
<pool name="Instruments">
  <choice name="1"><inst>Vocals</inst></choice>
  <choice name="2"><inst>Guitar</inst></choice>
  <choice name="3"><inst>Bass Guitar</inst></choice>
  <choice name="4"><inst>Drums</inst></choice>
  <choice name="5"><inst>Keyboards</inst></choice>
  <choice name="6"><inst>Banjo</inst></choice>
  <choice name="7"><inst>Violin</inst></choice>
  <choice name="8"><inst>Cello</inst></choice>
  <choice name="9"><inst>Trumpet</inst></choice>
  <choice name="10"><inst>Saxophone</inst></choice>
  <choice name="11"><inst>Clarinet</inst></choice>
  <choice name="12"><inst>Trombone</inst></choice>
</pool>
```
C.2 Genre

The Genre table contains IDs and names for all musical genres supported in the Hallux database.

C.2.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genre_ID</td>
<td>Unique id for the genre (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Genre</td>
<td>The actual name of the genre</td>
<td>CHAR(50)</td>
</tr>
</tbody>
</table>

C.2.2 Rules

None.

C.2.3 SDDL Description

```xml
<choice name="13"><inst>Flute</inst></choice>
<choice name="14"><inst>Tambourine</inst></choice>
<choice name="15"><inst>Harmonica</inst></choice>
</pool>

<table name="Instrument">
  <field name="Instrument_Id" type="int">
    <iteration pool="Instruments"/>
  </field>
  <field name="Instrument_Name" type="CHAR(50)">
    <formula>Instruments['"'+Instrument_Id]+'.inst'</formula>
  </field>
</table>

<pool name="GenreInstrument">
  <choice name="1">
    <name>Rock</name>
    <weight>5</weight>
  </pool>
  <choice name="2">
    <name>Guitar</name>
    <weight>5</weight>
  </choice>
  <choice name="3">
    <name>Bass Guitar</name>
    <weight>5</weight>
  </choice>
  <choice name="4">
    <name>Drums</name>
    <weight>5</weight>
  </choice>
  <choice name="5">
    <name>Keyboards</name>
    <weight>5</weight>
  </choice>
  <choice name="14">
    <name>Tamborine</name>
    <weight>5</weight>
  </choice>
</pool>
```
</pool>
</choice>
<choice name="2">
  <name>Pop</name>
  <weight>5</weight>
  <pool name="Instruments">
    <choice name="1"/> <!-- Vocals-->
    <choice name="2"/> <!-- Guitar-->
    <choice name="3"/> <!-- Bass Guitar-->
    <choice name="4"/> <!-- Drums-->
    <choice name="5"/> <!-- Keyboards-->
    <choice name="14"/> <!-- Tambourine-->
  </pool>
</choice>

<choice name="3">
  <name>Rap</name>
  <weight>5</weight>
  <pool name="Instruments">
    <choice name="1"/> <!-- Vocals-->
    <choice name="3"/> <!-- Bass Guitar-->
    <choice name="4"/> <!-- Drums-->
    <choice name="5"/> <!-- Keyboards-->
  </pool>
</choice>

<choice name="4">
  <name>Country</name>
  <weight>5</weight>
  <pool name="Instruments">
    <choice name="1"/> <!-- Vocals-->
    <choice name="2"/> <!-- Guitar-->
    <choice name="3"/> <!-- Bass Guitar-->
    <choice name="4"/> <!-- Drums-->
    <choice name="5"/> <!-- Keyboards-->
    <choice name="6"/> <!-- Banjo-->
    <choice name="7"/> <!-- Violin-->
    <choice name="15"/> <!-- Harmonica-->
  </pool>
</choice>

<choice name="5">
  <name>Jazz</name>
  <weight>2</weight>
  <pool name="Instruments">
    <choice name="1"/> <!-- Vocals-->
    <choice name="2"/> <!-- Guitar-->
    <choice name="3"/> <!-- Bass Guitar-->
    <choice name="4"/> <!-- Drums-->
    <choice name="5"/> <!-- Keyboards-->
    <choice name="9"/> <!-- Trumpet-->
    <choice name="10"/> <!-- Saxophone-->
    <choice name="11"/> <!-- Clarinet-->
    <choice name="12"/> <!-- Trombone-->
    <choice name="13"/> <!-- Flute-->
  </pool>
</choice>

<choice name="6">
  <name>Gospel</name>
  <weight>1</weight>
</choice>
<pool name="Instruments">
  <choice name="1"/> <!--Vocals-->
  <choice name="2"/> <!--Guitar-->
  <choice name="3"/> <!--Bass Guitar-->
  <choice name="4"/> <!--Drums-->
  <choice name="5"/> <!--Keyboards-->
</pool>
</choice>
<choice name="7">
  <name>Classical</name>
  <weight>2</weight>
  <pool name="Instruments">
    <choice name="4"/> <!--Drums-->
    <choice name="5"/> <!--Keyboards-->
    <choice name="7"/> <!--Violin-->
    <choice name="8"/> <!--Cello-->
    <choice name="9"/> <!--Trumpet-->
    <choice name="10"/> <!--Saxophone-->
    <choice name="11"/> <!--Clarinet-->
    <choice name="12"/> <!--Trombone-->
    <choice name="13"/> <!--Flute-->
  </pool>
</choice>
</pool>
<table name="Genre">
  <field name="Genre_ID" type="int">
    <iteration pool="GenreInstrument"/>
  </field>
  <field name="Genre" type="CHAR(50)">
    <formula>GenreInstrument["\""+Genre_ID\""].name</formula>
  </field>
</table>

Note that each genre in the GenreInstrument pool is associated with a set of instruments. This association is used later when constructing the Member_Instrument table that associates band members with musical instruments.

C.3 Zip_Code

The Zip_Code table contains state, city, latitude and longitude information for each zip code in the U.S., according to the 2000 Census.

C.3.1 Schema
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zip_Code</td>
<td>Unique United States ZIP code (PK)</td>
<td>CHAR(5)</td>
</tr>
<tr>
<td>State_Abbr</td>
<td>2-letter abbreviation for the enclosing state (FK)</td>
<td>CHAR(2)</td>
</tr>
<tr>
<td>City</td>
<td>The city the ZIP code represents</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>Latitude</td>
<td>The latitude of the ZIP code location</td>
<td>DECIMAL(10,7)</td>
</tr>
<tr>
<td>Longitude</td>
<td>The longitude of the ZIP code location</td>
<td>DECIMAL(10,7)</td>
</tr>
</tbody>
</table>

C.3.2 Rules

This set should be a realistic set of zip codes along with their associated states, cities, latitudes and longitudes.

C.3.3 SDDL Description

```xml
<import filename="StateZip.xml"/>
<import filename="ZipLatLon.xml"/>
<table name="Zip_Code">
  <variable name="st1" type="CHAR(2)">
    <iteration pool="StateZip"/>
  </variable>
  <variable name="zip1" type="string">
    <iteration pool="StateZip[st1].zips"/>
  </variable>
  <field name="Zip_Code" type="CHAR(5)">
    <formula>FILL(zip1,5,0-1,"0")</formula>
  </field>
  <field name="State_Abbr" type="CHAR(2)">
    <formula>st1</formula>
  </field>
  <field name="City" type="CHAR(50)">
    <formula>StateZip[st1].zips[zip1].city</formula>
  </field>
  <field name="Latitude" type="DECIMAL(10,7)">
    <formula>0.0+ZipLatLon[zip1].lat</formula>
  </field>
  <field name="Longitude" type="DECIMAL(10,7)">
    <formula>0.0+ZipLatLon[zip1].lon</formula>
  </field>
</table>
```

This table is generated by iterating though all zip codes in the StateZip pool, and gathering associated lat/lon information from the ZipLatLon pool.
C.4 State

The State table associates state abbreviations with state names.

C.4.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>State_Abbr</td>
<td>2-letter abbreviation for the state (PK)</td>
<td>CHAR(2)</td>
</tr>
<tr>
<td>State_Name</td>
<td>The full name of the state represented by the abbreviation</td>
<td>CHAR(20)</td>
</tr>
</tbody>
</table>

C.4.2 Rules

None.

C.4.3 SDDL Description

```xml
<import filename="StateAbbrToName.xml"/>
<table name="State">
  <field name="State_Abbr" type="CHAR(2)">
    <iteration pool="StateAbbrToName"/>
  </field>
  <field name="State_Name" type="CHAR(20)">
    <formula>StateAbbrToName[State_Abbr].full</formula>
  </field>
</table>
```

This table is generated by iterating through the StateAbbrToName pool. There are some entries generated for territories (i.e., “U.S. Virgin Islands”) that are not referenced in the Zip_Code table. However, all states referenced in the Zip_Code table are included in this table.

C.5 Venue

The Venue table holds information about venues at which Hallux bands could perform.
### C.5.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Venue_ID</td>
<td>Unique id for the venue (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Venue_Name</td>
<td>Name of the venue</td>
<td>CHAR(100)</td>
</tr>
<tr>
<td>Street_Address</td>
<td>The street address for the venue</td>
<td>CHAR(100)</td>
</tr>
<tr>
<td>ZIP_Code</td>
<td>ZIP code for the location of the venue (FK)</td>
<td>CHAR(5)</td>
</tr>
<tr>
<td>ZIP_Code_Ext</td>
<td>The last 4 digits of an extended 9 digit ZIP code</td>
<td>CHAR(4)</td>
</tr>
<tr>
<td>Contact_Phone</td>
<td>Phone number for a contact at this venue</td>
<td>CHAR(20)</td>
</tr>
</tbody>
</table>

### C.5.2 Rules

There should be 100 venues, in random locations. Venue_Name values should denote small club environments. Zip_Code_Ext is optional.

### C.5.3 SDDL Description

```xml
<import filename="StateZip.xml"/>
<import filename="names.xml"/>
<import filename="streetnames.xml"/>

<pool name="venuetypes">
  <choice name="Bar"/>
  <choice name="Bar and Grill"/>
  <choice name="Club"/>
</pool>

<table name="Venue">
  <field name="Venue_Id" type="int">
    <iteration base="1000" count="100">
      <itervar name="state" type="string" init="StateZip"/>
      <itervar name="zip" type="string" init="StateZip[state].zips"/>
      <itervar name="eth" type="string" init="names"/>
      <itervar name="fname" type="string" init="names[eth].firstnames"/>
      <itervar name="street" type="string" init="streetnames"/>
      <itervar name="stype" type="string" init="streettypes"/>
    </iteration>
  </field>
  <variable name="namechooser" type="int">
    <formula>IRND(4)</formula>
  </variable>
  <field name="Venue_Name" type="CHAR(100)">
    <formula>namechooser==0?(fname+'''s'):
      namechooser==1?(fname+'''s on '+street):
      namechooser==2?(fname+'''s near '+street):
      namechooser==3?(fname+'''s at '+street):
      namechooser==4?(fname+'''s on '+street)+'</formula>
  </field>
</table>
```
Note the following:

- The double-apostrophe notation is interpreted by most SQLs as a single apostrophe. It is used to escape apostrophes in single-quoted values.
- `snum` represents “street number” and `sdir` represents “street direction.”

### C.6 Producer

The **Producer** table contains IDs and names for producers of Hallux videos.

#### C.6.1 Schema
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer_ID</td>
<td>Unique id for the producer (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Producer_Name</td>
<td>Name of the video producing company</td>
<td>CHAR(100)</td>
</tr>
</tbody>
</table>

### C.6.2 Rules

This table needs to contain information for 10-12 video producers.

### C.6.3 SDDL Description

```xml
<pool name="ProducerNames">
  <choice name="XYZ">
    <name>ABC Records</name>
    <id>1</id>
  </choice>
  <choice name="Acme">
    <name>Acme Productions</name>
    <id>2</id>
  </choice>
  <choice name="Zebra">
    <name>Zebra Music</name>
    <id>3</id>
  </choice>
  <choice name="HoneyBear">
    <name>HoneyBear Records</name>
    <id>4</id>
  </choice>
  <choice name="TooCool">
    <name>2Cool4U Music</name>
    <id>5</id>
  </choice>
  <choice name="SheHateMe">
    <name>SheHateMe Group</name>
    <id>6</id>
  </choice>
  <choice name="CroMagnon">
    <name>CroMagnon Group</name>
    <id>8</id>
  </choice>
  <choice name="Hitz">
    <name>Hitz Records</name>
    <id>9</id>
  </choice>
  <choice name="AllStar">
    <name>All Star Music</name>
    <id>10</id>
  </choice>
  <choice name="Thuggery">
    <name>Thuggery Music</name>
    <id>11</id>
  </choice>
</pool>
```
C.7 Order_Source

The Order_Source table contains information about possible sources of Hallux orders.

C.7.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order_Source_ID</td>
<td>Unique id for the source (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Source_Name</td>
<td>Name of the source (T-Jams, phone, online, etc)</td>
<td>CHAR(100)</td>
</tr>
<tr>
<td>Source_Description</td>
<td>Short description of the source</td>
<td>CHAR(200)</td>
</tr>
</tbody>
</table>

C.7.2 Rules

There should be 4 order sources: Phone, Fax, Online, T-Jams.

C.7.3 SDDL Description

```xml
<pool name="sources">
  <choice name="1">
    <nm>Phone</nm>
    <ds>Order taken by phone</ds>
  </choice>
  <choice name="2">
    <nm>Fax</nm>
    <ds>Order received by fax</ds>
  </choice>
</pool>
```
The `Order_Source` table is constructed by iterating through the `sources` pool.

C.8 Person

The `Person` table contains personal information about all Hallux band members and agents. It will later be used to construct the `Band_Member` and `Agent` tables.

C.8.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person_ID</td>
<td>Unique id for person (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>First_Name</td>
<td>The person's first name</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>Last_Name</td>
<td>The person's last name</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>Street_Address</td>
<td>The street address where the person lives</td>
<td>CHAR(100)</td>
</tr>
<tr>
<td>ZIP_Code</td>
<td>The ZIP code of the city where the person lives (FK)</td>
<td>CHAR(5)</td>
</tr>
<tr>
<td>ZIP_Code_Ext</td>
<td>The last 4 digits of an extended 9 digit ZIP code</td>
<td>CHAR(4)</td>
</tr>
<tr>
<td>Phone_Number</td>
<td>The person's phone number</td>
<td>CHAR(20)</td>
</tr>
</tbody>
</table>
C.8.2 Rules

There are 400 bands, each with 1-6 members. There are also 42 agents. Some bands are composed of members from the same state, some are not.

C.8.3 SDDL Description

The Person table was actually constructed from two SDDL files: “hallux_PersonBM.xml” for band members, and “hallux_PersonA.xml” for agents.

C.8.3.1 hallux_PersonBM.xml

```
<seed>38578137</seed>
<import filename="StateZip.xml"/>
<import filename="names.xml"/>
<import filename="streetnames.xml"/>
<pool name="email_domains">
  <choice name="juno.com"/>
  <choice name="netzero.com"/>
  <choice name="hotmail.com"/>
  <choice name="gmail.com"/>
  <choice name="yahoo.com"/>
  <choice name="cox.com"/>
  <choice name="sbcglobal.com"/>
</pool>
<table name="Person">
  <variable name="BandID" type="int">
    <iteration base="1000" count="400">
      <itervar name="refState" type="string" init="StateZip"/>
      <itervar name="sameState" type="bool" init="IRND(3)&gt;0"/>
    </iteration>
  </variable>
  <variable name="index" type="int">
    <iteration base="0" count="1+IRND(6)"/>
  </variable>
  <field name="Person_Id" type="int">
    <formula>100000+(BandID*10)+index</formula>
  </field>
  <variable name="ethnicity" type="string">
    <formula>names</formula>
  </variable>
  <field name="First_Name" type="CHAR(50)">
    <formula>names[ethnicity].firstnames</formula>
  </field>
  <field name="Last_Name" type="CHAR(50)">
```
Note that the Band_ID iteration counts 400 bands, and the index iteration counts 1-6 members per band. Person_ID is constructed by multiplying Band_ID by 10, adding index, and adding to 100000. The sameState iteration variable has a 2-in-3 chance of being true. If sameState is true, then all members of that band will be from refState; otherwise, members will be from randomly chosen states.

C.8.3.2 Hallux_PersonA.xml

<seed>875548975</seed>
<import filename="StateZip.xml"/>
<import filename="names.xml"/>
<import filename="streetnames.xml"/>
<pool name="email_domains">
    <choice name="juno.com"/>
    <choice name="netzero.com"/>
    <choice name="hotmail.com"/>
    <choice name="gmail.com"/>
    <choice name="yahoo.com"/>
    <choice name="cox.com"/>
    <choice name="sbcglobal.com"/>
</pool>
<table name="Person">
    <variable name="index" type="int">
        <iteration base="1" count="42"/>
    </variable>
    <field name="Person_Id" type="int">
        <formula>200000+index</formula>
    </field>
    <variable name="ethnicity" type="string">
        <formula>names</formula>
    </variable>
    <field name="First_Name" type="CHAR(50)">
        <formula>names[ethnicity].firstnames</formula>
    </field>
    <field name="Last_Name" type="CHAR(50)">
        <formula>names[ethnicity].lastnames</formula>
    </field>
    <variable name="snum" type="int">
        <dist>
            <tier prob="0.15" min="10" max="99"/>
            <tier prob="0.25" min="100" max="999"/>
            <tier prob="0.5" min="1000" max="9999"/>
            <tier prob="0.1" min="10000" max="20000"/>
        </dist>
    </variable>
    <variable name="sdir" type="string">
        <formula>(IRND(10)&lt;5)?f":directions":" </formula>
    </variable>
    <field name="Street_Address" type="CHAR(100)">
        <formula>"+snum+" +sdir+streetnames+" +streettypes</formula>
    </field>
    <variable name="st2" type="string">
        <formula>StateZip</formula>
    </variable>
    <field name="Zip_Code" type="CHAR(5)">
        <formula>FILL((''+StateZip[st2].zips),5,0-1,"0")</formula>
    </field>
    <field name="Zip_Code_Ext" type="CHAR(4)">
        <formula>IRND(3)==1?FILL(IRND(10000),4,0-1,"0"):"</formula>
    </field>
    <field name="Phone_Number" type="CHAR(20)">
        <formula>FILL((200+IRND(800)),3,0-1,"0")+
            FILL(IRND(1000),3,0-1,"0")+
            FILL(IRND(10000),4,0-1,"0")</formula>
    </field>
</table>
Note that the “Agent” portion of the SDDL description is very similar to the “Band Member” portion; they must of necessity have the same schema, and most of the logic is very similar. For this table, though, only 42 entries are generated (one for each agent), and Person_ID is constructed by adding 200000 to the agent index. Note also that a different random seed was used for the “Agent” portion of the table, to avoid repeating the addresses from the “Band Member” portion of the table.

C.9 Agent

The Agent table contains business data associated with each agent.

C.9.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent_ID</td>
<td>Unique id for an agent (PK,FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Hire_Date</td>
<td>The date Hallux hired the agent</td>
<td>DATE</td>
</tr>
<tr>
<td>Agent_Status_Code</td>
<td>Active or inactive</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>Commission</td>
<td>The percentage of sales this agent receives</td>
<td>DECIMAL(5,4)</td>
</tr>
<tr>
<td>Salary</td>
<td>This agent's salary</td>
<td>DECIMAL(18,2)</td>
</tr>
</tbody>
</table>

C.9.2 Rules

Agent_ID is the same as the Person.Person_ID value for the agent. One-tenth of Hallux agents were hired at the inception of Hallux (January 1, 1990), the rest were hired somewhere in the 1990s.
C.9.3 SDDL Description

<table name="Agent">
   <field name="Agent_Id" type="int">
      <iteration query="select Person_Id from Person
         where Person_Id &gt;= 200000
         order by Person_Id"/>
   </field>
   <field name="Hire_Date" type="date">
      <formula>IRND(10)==0
         ? #1990-01-01#
         : #1990-01-01#+IRND(3650)</formula>
   </field>
   <field name="Agent_Status_Code" type="CHAR(1)">
      <formula>"A"</formula>
   </field>
   <field name="Commission" type="DECIMAL(5,4)">
      <min>0.04</min>
      <max>0.10</max>
   </field>
   <field name="Salary" type="DECIMAL(18,2)">
      <min>20000.0</min>
      <max>50000.0</max>
   </field>
</table>

The Agent table is constructed by iterating through all rows in the Person table that have a Person_Id &gt;= 200000.

C.10 Band

The Band table contains information about each Hallux band.

C.10.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Band_ID</td>
<td>Unique id for the band</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Primary.Contact_ID</td>
<td>A Person_ID, the primary contact for the band (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Band_Name</td>
<td>The name of the band</td>
<td>CHAR(100)</td>
</tr>
<tr>
<td>ZIP_Code</td>
<td>The ZIP code of the city where the band is located (FK)</td>
<td>CHAR(5)</td>
</tr>
<tr>
<td>Band_Status_Code</td>
<td>Active or inactive</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>Formation_Date</td>
<td>The date the band officially formed</td>
<td>DATE</td>
</tr>
</tbody>
</table>
C.10.2 Rules

The Primary Contact ID value can reference either a member of the band or an agent. One-tenth of the bands were formed at Hallux inception (January 1, 1990) and the rest were formed some time between then and December 31, 2006.

C.10.3 SDDL Description

```xml
<import filename="Words.xml"/>
<table name="Band">
  <variable name="query" type="string">
    <iteration query="select person_id, first_name, last_name, zip_code from Person where person_id &lt; 200000 and person_id mod 10 = 0 order by person_id"/>
  </variable>
  <variable name="pmin" type="int">
    <formula>0+query.person_id</formula>
  </variable>
  <variable name="pmax" type="int">
    <formula>0+query.person_id+10</formula> <!--MAGIC NUMBER!!-->
  </variable>
  <variable name="bandcount" type="int">
    <queryPool query="select count(*) from Person where person_id &gt;= [pmin] and person_id &lt; [pmax]"/>
  </variable>
  <variable name="fname" type="string">
    <formula>SPLIT(""+query.first_name," ",0)</formula>
  </variable>
  <variable name="lname" type="string">
    <formula>SPLIT(""+query.last_name," ",0)</formula>
  </variable>
  <field name="Band_Id" type="int">
    <formula>(0+query.person_id-100000)/10</formula>
  </field>
  <variable name="agentPool" type="int">
    <queryPool query="select agent_id from Agent order by agent_id"/>
  </variable>
  <field name="Primary Contact_Id" type="int">
    <formula>IRND(2)==0 ? (0+query.person_id) : (0+agentPool)</formula>
  </field>
  <variable name="fmt_chooser" type="int">
    <min>1</min>
</table>
```
There is a numerical correlation between a band member's `person_id` and the `band_id` of their band. For any band, the band members have `person_ids` ranging from \((100000+\text{band}_id*10)\) to \((100000+\text{band}_id*10+5)\), with the range depending upon the number of members in the band. Thus, a band will always have at least one member, and so a member with a `person_id` of \((100000+\text{band}_id*10)\). The query iteration will thus cycle through the first member of every band.

The `pmin`, `pmax` and `bandcount` variables will determine the number of members in the band. This information is important when choosing a band name. If
there is only one member, then the band name will be set to the name of that member; otherwise, the band name will be synthesized according to band-naming rules.

C.11 Contract

The Contract table contains contract information associated with each Hallux band.

C.11.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contract_ID</td>
<td>Unique id for the contract (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Band_ID</td>
<td>Identifies the band on this contract (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Agent_ID</td>
<td>Identifies the agent on this contract (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Begin_Date</td>
<td>Optional: The date a contract goes into effect</td>
<td>DATE</td>
</tr>
<tr>
<td>End_Date</td>
<td>The date a contract ends. Required if there is a Begin_Date, Null if not</td>
<td>DATE</td>
</tr>
<tr>
<td>Album_Rev_Pct</td>
<td>Percentage the band gets from sales</td>
<td>DECIMAL(18,2)</td>
</tr>
<tr>
<td>Live_Rev_Pct</td>
<td>Percentage the band gets from a live performance</td>
<td>DECIMAL(18,2)</td>
</tr>
<tr>
<td>Album_Count</td>
<td>Optional: The number of albums to be produced under this contract</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>Live_Count</td>
<td>Optional: The number of live performances under this contract</td>
<td>SMALLINT</td>
</tr>
</tbody>
</table>

C.11.2 Rules

For now, there is one contract per band. Each contract can be defined in one of three ways: Begin_Date/End_Date, Album_Count/Album_Rev_Pct, or Live_Count/Live_Rev_Pct. A band’s take is typically 5-15% of album revenue and 40-50% of live performance revenue.

C.11.3 SDDL Description
<table name="Contract">
  <field name="Band_Id" type="int">
    <iteration query="select Band_Id, formation_date, 
                  primary_contact_id
      from Band
      order by Band_Id"/>
  </field>
  <field name="Contract_Id" type="int">
    <formula>Band_Id*10</formula>
  </field>
  <variable name="form_date" type="date">
    <formula>DATE(Band_Id$formation_date)</formula>
  </variable>
  <variable name="Agent_Pool" type="int">
    <queryPool query="select agent_id from Agent
      where hire_date &lt;= '[form_date]' 
      order by agent_id"/>
  </variable>
  <field name="Agent_Id" type="int">
    <formula>((0+Band_Id$primary_contact_id)&gt;200000) 
      ? (0+Band_Id$primary_contact_id) : Agent_Pool</formula>
  </field>
  <variable name="contract_type" type="int">
    <min>1</min>
    <max>3</max>
  </variable>
  <field name="Begin_Date" type="date">
    <formula>contract_type==1 ?
      (form_date+IRND(365)) : "null"</formula>
  </field>
  <field name="End_Date" type="date">
    <formula>contract_type==1? (#2010-01-01#):"null"</formula>
  </field>
  <field name="Album_Rev_Pct" type="DECIMAL(18,2)">
    <formula>0.05+(IRND(11)/100.0)</formula>
  </field>
  <field name="Album_Count" type="int">
    <formula>contract_type==2?(1+IRND(5)):"null"</formula>
  </field>
  <field name="Live_Rev_Pct" type="DECIMAL(18,2)">
    <formula>0.40+(IRND(11)/100.0)</formula>
  </field>
  <field name="Live_Count" type="int">
    <formula>contract_type==3?(10+IRND(91)):"null"</formula>
  </field>
</table>

C.12 Band_Genre

The Band_Genre table associates each band with one or more musical genres.
C.12.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Band_Id</td>
<td>The id number for the band this member plays in (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Genre_Id</td>
<td>The id of a genre associated with this band (FK)</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

C.12.2 Rules

There should be one or two Genre_Ids for each Band_Id.

C.12.3 SDDL Description

```xml
<import filename="hallux_GenreInstrumentPool.xml"/>
<table name="Band_Genre">
  <field name="Band_Id" type="int">
    <iteration query="select band_id from Band
                     order by band_id">
      <repeatDist>
        <tier prob="0.5" min="1" max="1"/>
        <tier prob="0.5" min="2" max="2"/>
      </repeatDist>
      <itervar name="firstGenre" type="string"
               init=""NOTHING"quot;quot;"/>
    </iteration>
  </field>
  <variable name="genreTry" type="string">
    <formula>GenreInstrument</formula>
  </variable>
  <field name="Genre_Id" type="int">
    <formula>(genreTry==firstGenre) ?
               GenreInstrument[genreTry]+(1+IRND(5)) :
               genreTry</formula>
  </field>
  <variable name="updateFirstGenre" type="string">
    <formula>firstGenre=""+Genre_Id</formula>
  </variable>
</table>
```

The Band_Id field iterates through the bands in the Band table, with a repeat count of 1 or 2. The genreTry variable selects a genre, and the Genre_Id field assures that the Genre_Id selected is not a repeat value for the band.

The GenreInstrument pool is listed in section C.2.3.
C.13 Band_Member

The Band_Member table contains information about every member of every Hallux band.

C.13.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member_ID</td>
<td>Uniquely Identifies a Band Member (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Band_ID</td>
<td>The id number for the band this member plays in (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Join_Date</td>
<td>The date this band member joined the band</td>
<td>DATE</td>
</tr>
<tr>
<td>Member_Status_Code</td>
<td>Active member or inactive member of the band</td>
<td>CHAR(1)</td>
</tr>
</tbody>
</table>

C.13.2 Rules

The Join_Date for a member has a 90% chance of being the band’s formation date, and a 10% chance of being some time between the band’s formation date and December 31, 2006. The Member_ID is the same as the Person.Person_ID value for a band member.

C.13.3 SDDL Description

```xml
<table name="Band_Member">
  <field name="Member_Id" type="int">
    <iteration query="select person_id, band_id, formation_date from Person, Band where Band.band_id = floor((Person.person_id-100000)/10) order by person_id"/>
  </field>
  <field name="Band_Id" type="int">
    <formula>Member_Id$band_id</formula>
  </field>
  <variable name="form_date" type="date">
    <formula>Member_Id$formation_date</formula>
  </variable>
  <field name="Join_Date" type="date">
```
C.14 Member_Instrument

The Member_Instrument table contains information about the instrument(s)
played by each member of each Hallux band.

C.14.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member_Id</td>
<td>Band member ID (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Instrument_Id</td>
<td>The id of the instrument played by member (FK)</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

C.14.2 Rules

Each band member should play one or two instruments. The instruments should
be common to the genre(s) associated with the band.

C.14.3 SDDL Description

```
<import filename="hallux_GenreInstrumentPool.xml"/>
<table name="Member_Instrument">
  <field name="Member_Id" type="int">
    <iteration query="select member_id, band_id
                     from Band_Member
                     order by member_id, band_id">
      <repeatDist>
        <tier prob="0.4" min="1" max="1"/>
        <tier prob="0.6" min="2" max="2"/>
      </repeatDist>
      <itervar name="firstInst" type="string">
        <init>"nothing"</init>
      </itervar>
    </iteration>
  </field>
</table>
```
The table is constructed by iterating through all rows of the Band_Member table. Each band member has a 40% chance of playing a single instrument, and a 60% chance of playing 2 instruments. The instTry logic helps to assure that a band member will not be assigned the same instrument twice. The genre variable selects a genre associated with the band member’s band, and instruments are chosen that are common to that genre.

The GenreInstrument pool is listed in section C.2.3.

C.15 Album

The Album table contains information about each album produced by each Hallux band.

C.15.1 Schema
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Album_ID</td>
<td>Unique id for the album (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Band_ID</td>
<td>The id of the band that made the album (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Album_Name</td>
<td>The name of the album</td>
<td>CHAR(100)</td>
</tr>
<tr>
<td>Release_Date</td>
<td>The date the album was released</td>
<td>DATE</td>
</tr>
<tr>
<td>Production_Cost</td>
<td>The cost of making the album</td>
<td>DECIMAL(18,2)</td>
</tr>
</tbody>
</table>

C.15.2 Rules

Bands will release albums every 12-18 months, with at least a year between albums. The Release_Date for a band’s first album will be the same as the band’s formation date. The Production_Cost value has a 10% chance of being between $1000 and $5000, a 50% chance of being between $5000 and $10000, a 30% chance of being between $10000 and $20000, and a 10% chance of being between $20000 and $60000.

C.15.3 SDDL Description

```
<import filename="Words.xml"/>
<table name="Album">
  <field name="Band_Id" type="int">
    <iteration query="select Band_Id, Formation_Date from Band order by Band_Id"/>
  </field>
  <variable name="AlbIdx" type="int">
    <iteration base="0" count="((#2006-12-31#-DATE(Band_Id$Formation_Date))/550)+1"/>
  </variable>
  <field name="Album_Id" type="int">
    <formula>Band_Id*20+AlbIdx</formula>
  </field>
  <variable name="nameFmt" type="int">
    <dist>
      <tier prob="0.140" min="1" max="1"/>
      <tier prob="0.140" min="2" max="2"/>
      <tier prob="0.140" min="3" max="3"/>
      <tier prob="0.010" min="4" max="4"/>
      <tier prob="0.010" min="5" max="5"/>
      <tier prob="0.140" min="6" max="6"/>
      <tier prob="0.140" min="7" max="7"/>
      <tier prob="0.140" min="8" max="8"/>
    </dist>
  </variable>
</table>
```
This table is constructed by iterating through the Band table. For each band, the number of albums produced (i.e., the count for the AlbIdx iteration) is defined by

\[
\left( \frac{\text{number of days between band formation and December 31, 2006}}{540} \right) + 1.
\]

The nameFmt variable chooses a name format, which is enforced by the Album_Name field. The Album_Id assigned to each album is \((\text{Band_Id} \times 20 + \text{AlbIdx})\).
The Song table contains information about each song from each Hallux album.

C.16.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Song_ID</td>
<td>Unique id for the song (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Album_ID</td>
<td>The album the song is on (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Song_Name</td>
<td>The name of the song</td>
<td>CHAR(100)</td>
</tr>
<tr>
<td>Sequence</td>
<td>The track of the song on the album</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>Duration_Seconds</td>
<td>The length of the song in seconds</td>
<td>SMALLINT</td>
</tr>
</tbody>
</table>

C.16.2 Rules

There should be 6-15 songs per album. Each song has an 85% chance of being between 2:40 and 3:20 in length, and a 15% chance of being between 5:00 and 10:00 in length.

C.16.3 SDDL Description

```xml
<import filename="Words.xml"/>
<import filename="names.xml"/>
<import filename="StateZipUW.xml"/>  
<import filename="StateAbbrToName.xml"/>
<table name="Song">
    <field name="Album_Id" type="int">  
        <iteration query="select Album_Id from Album
                        order by Album_Id"/>
    </field>
    <field name="Sequence" type="int">  
        <iteration base="1" count="6+IRND(10)"/>
    </field>
    <field name="Song_Id" type="int">  
        <formula>Album_Id*20+Sequence</formula>
    </field>
    <variable name="state" type="string">  
        <formula>StateZip</formula>
    </variable>
    <variable name="zip" type="string">  
        <formula>StateZip[state].zips</formula>
    </variable>
    <variable name="city" type="string">  
        <formula>StateZip[state].zips[zip].city</formula>
    </variable>
    <variable name="statefull" type="string">  
        <formula>StateAbbrToName[state].full</formula>
</table>
```
Most of the logic here has to do with choosing a name for a song. The nameFmt variable chooses one of nine name formats, and the Song_Name field enforces that format.

C.17 Video

The Video table contains information about each video sold by Hallux Productions.

C.17.1 Schema
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video_ID</td>
<td>Unique id for the video (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Video_Name</td>
<td>The title of the video</td>
<td>CHAR(100)</td>
</tr>
<tr>
<td>Producer_ID</td>
<td>The producing company that produced this video (FK)</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

C.17.2 Rules

There should be 200 videos in the table.

C.17.3 SDDL Description

```xml
<import filename="hallux_GenreInstrumentPool.xml"/>
<import filename="StateZip.xml"/>
<import filename="StateAbbrToName.xml"/>
<pool name="video_nouns">
  <choice name="Favorites"/>
  <choice name="Videos"/>
  <choice name="Concerts"/>
  <choice name="Hits"/>
</pool>
<pool name="regions">
  <choice name="The South"/>
  <choice name="The West Coast"/>
  <choice name="The Pacific Northwest"/>
  <choice name="The Gulf Coast"/>
  <choice name="The East Coast"/>
  <choice name="New England"/>
  <choice name="The Midwest"/>
</pool>
<table name="Video">
  <field name="Video_ID" type="int">
    <iteration base="50000" count="200">
      <itervar name="gnum" type="string" init="GenreInstrument"/>
      <itervar name="genre" type="string" init="GenreInstrument[gnum].name"/>
      <itervar name="instNum" type="string" init="Instruments"/>
      <itervar name="inst" type="string" init="Instruments[instNum].inst"/>
      <itervar name="state" type="string" init="StateZip"/>
      <itervar name="zip" type="string" init="StateZip[state].zips"/>
      <itervar name="city" type="string" init="StateZip[state].zips[zip].city"/>
      <itervar name="fullstate" type="string" init="StateAbbrToName[state].full"/>
    </iteration>
  </field>
  <variable name="placenameSelector" type="int">
```

321
Again, this is a fairly simple table, with most of the logic devoted to picking a realistic random name for each video. The subject variable is either a musical genre name or an instrument name. The placename variable is a city, a state, or a region. The video_nouns pool contains the entries “Favorites”, “Videos”, “Concerts” and “Hits”. The name is formed by concatenating randomly chosen subject, video_noun, placename values.

C.18 Item_Type

The Item_Type table is a support table for the Item table, and contains information about each of 3 item types.
C.18.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item_Type_ID</td>
<td>Unique id for each type (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Item_Type</td>
<td>Name of a type, either Album, Song, or Video</td>
<td>CHAR(20)</td>
</tr>
</tbody>
</table>

C.18.2 Rules

This table should support item types “Album”, “Song” (single), and “Video”.

C.18.3 SDDL Description

```xml
<pool name="ItemTypes">
  <choice name="1"><ds>Album</ds></choice>
  <choice name="2"><ds>Song</ds></choice>
  <choice name="3"><ds>Video</ds></choice>
</pool>

<table name="Item_Type">
  <field name="Item_Type_ID" type="int">
    <iteration pool="ItemTypes"/>
  </field>
  <field name="Item_Type" type="CHAR(20)">
    <formula>ItemTypes[""+Item_Type_ID].ds</formula>
  </field>
</table>
```

C.19 Item

The Item table has an entry for each album, song and video sold by Hallux Productions.

C.19.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item_ID</td>
<td>This is a Song_ID, Album_ID, or Video_ID (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Item_Type_ID</td>
<td>Connects this item to an item type (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Item_Description</td>
<td>Short description of the item</td>
<td>CHAR(200)</td>
</tr>
</tbody>
</table>
C.19.2 Rules

Include all albums, songs and videos in this table.

C.19.3 SDDL Description

The Item table is actually generated in three segments, each of which has its own SDDL description: Album items, Song items, and Video items.

The album item portion is generated as follows:

```xml
<table name="Item">
  <field name="Item_ID" type="int">
    <iteration query="select album_id,album_name,band_name from Album,Band where Album.band_id = Band.band_id order by album_id"/>
  </field>
  <field name="Item_Type_ID" type="int">
    <formula>1</formula>
  </field>
  <field name="Item_Description" type="CHAR(200)">
    <formula>"Album: "+TRIM(Item_ID$band_name)+": "+TRIM(Item_ID$album_name)</formula>
  </field>
</table>
```

The song item portion is generated as follows:

```xml
<table name="Item">
  <field name="Item_ID" type="int">
    <iteration query="select song_id,song_name,album_name,band_name from Song,Album,Band where Song.album_id=Album.album_id and Album.band_id=Band.band_id order by song_id"/>
  </field>
  <field name="Item_Type_ID" type="int">
    <formula>2</formula>
  </field>
  <field name="Item_Description" type="CHAR(200)">
    <formula>"Song: "+TRIM(Item_ID$band_name)+": "+TRIM(Item_ID$album_name)+": "+TRIM(Item_ID$song_name)</formula>
  </field>
</table>
```
The video item portion is generated as follows:

```xml
<table name="Item">
    <field name="Item_ID" type="int">
        <iteration query="select video_id, video_name, producer_name from Video, Producer
                          where Video.producer_id = Producer.producer_id
                          order by video_id"/>
    </field>
    <field name="Item_Type_ID" type="int">
        <formula>3</formula>
    </field>
    <field name="Item_Description" type="CHAR(200)">
        <formula>"Video: "+TRIM(Item_ID$video_name)+
                     " ("+TRIM(Item_ID$producer_name)+
                     ")"</formula>
    </field>
</table>
```

IDs for Songs, Albums and Videos were carefully chosen previously so as not to conflict with each other in this table.

**C.20 Customer**

The Customer table contains information about each Hallux customer. A customer can be a corporate customer or an individual customer.

**C.20.1 Schema**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer_ID</td>
<td>Unique id for the customer</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Name</td>
<td>Customer's full name; Either a company name or the person's combined full name</td>
<td>CHAR(100)</td>
</tr>
<tr>
<td>Last_Name</td>
<td>Customer's last name if it is an individual person using T-Jams</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>First_Name</td>
<td>Customer's first name if it is an individual person using T-Jams</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>Street_Address</td>
<td>The customer's street address</td>
<td>CHAR(100)</td>
</tr>
<tr>
<td>ZIP_Code</td>
<td>The ZIP code of a person's location (FK)</td>
<td>CHAR(5)</td>
</tr>
<tr>
<td>ZIP_Code_Ext</td>
<td>The last 4 digits of an extended 9 digit ZIP code</td>
<td>CHAR(4)</td>
</tr>
</tbody>
</table>
C.20.2 Rules

There should be 5000 customers. About 10% of customers are corporate customers, the rest are individual customers.

C.20.3 SDDL Description

```xml
<import filename="StateZip.xml"/>
<import filename="names.xml"/>
<import filename="streetnames.xml"/>
<pool name="email_domains">
  <choice name="juno.com"/>
  <choice name="netzero.com"/>
  <choice name="hotmail.com"/>
  <choice name="gmail.com"/>
  <choice name="yahoo.com"/>
  <choice name="cox.com"/>
  <choice name="sbcglobal.com"/>
</pool>
<pool name="BusinessType">
  <choice name="Records"><dom>Records</dom></choice>
  <choice name="Media"><dom>Media</dom></choice>
  <choice name="Books"><dom>Books</dom></choice>
  <choice name="Book and Tape"><dom>BookNTape</dom></choice>
  <choice name="Discount Store"><dom>Discount</dom></choice>
</pool>
<table name="Customer">
  <variable name="cidx" type="int">
    <iteration base="1" count="5000">
      <itervar name="isPerson" type="bool" init="IRND(30000)&gt;3000"/>
      <itervar name="ethnicity" type="string" init="names"/>
      <itervar name="state" type="string" init="StateZip[state].zips"/>
      </iteration>
  </variable>
  <field name="Customer_Id" type="int">
    <formula>100000+(cidx*2)+(isPerson?0:1)</formula>
  </field>
  <field name="First_Name" type="CHAR(50)">
    <formula>isPerson ? ""+names[ethnicity].firstnames : ""</formula>
  </field>
  <field name="Last_Name" type="CHAR(50)">
    <formula>isPerson ? ""+names[ethnicity].lastnames : "(NONE)"</formula>
  </field>
</table>
```
<field name="Name" type="CHAR(100)">
    <formula>isPerson ?
        (First_Name + " " + Last_Name) :
        (busName1 + " " + busName2)</formula>
</field>

<field name="Street_Address" type="CHAR(100)">
    <formula>"" + snum + " " + sdir + streetnames + " " + streettypes</formula>
</field>

<field name="Zip_Code" type="CHAR(5)">
    <formula>FILL("" + zip),5,0-1,"0")</formula>
</field>

<field name="Zip_Code_Ext" type="CHAR(4)">
    <formula>IRND(3)==1?
        FILL(IRND(10000),4,0-1,"0") :
        ""</formula>
</field>

<field name="Phone_Number" type="CHAR(20)">
    <formula>FILL((200+IRND(800)),3,0-1,"0") +
        FILL(IRND(1000),3,0-1,"0") +
        FILL(IRND(10000),4,0-1,"0")</formula>
</field>

<field name="Email" type="CHAR(100)">
    <formula>isPerson ?
        (Last_Name + IRND(10000) + @ + email_domains):
        {"info@" + busName1 + BusinessType[busName2].dom + ".com"}
    </formula>
</field>

<field name="gender" type="CHAR(1)">
    <formula>isPerson ?
        "" + names[ethnicity].firstnames[First_Name].gender :
        ""
    </formula>
</field>
Corporate customers are given odd-numbered customer_ids, and individuals are given even-numbered customer_ids. This will help some later-generated tables to distinguish between corporate and individual customers. The gender field is added here, though it was not specified in the schema; it will be used in generating the Customer_Profile table, then will be removed.

C.21 Order_Detail

The Order_Detail table contains line item entries associated with the Order_Header table.

C.21.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order_ID</td>
<td>Unique order id from the Order_Header table (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Line_Number</td>
<td>Represents the line this item appears on the order (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Quantity</td>
<td>The quantity of this item ordered on this order</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Unit_Price</td>
<td>Price per unit of this item</td>
<td>DECIMAL(18,2)</td>
</tr>
<tr>
<td>Item_ID</td>
<td>Song_ID, Album_ID, or Video_ID from the Item table (FK)</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

C.21.2 Rules

The number of orders/day slowly rises from an average of 10 at Hallux inception (January 1, 1990) to an average of 50 at the end of 2006. Also, one cannot order a song or album if it has not yet been released.
Individuals order anywhere from 1-5 line items (usually 1) per order, with a quantity of 1 each. Individuals order 60% songs, 30% albums, and 10% videos. Individuals pay $0.95 for songs, and one of {$5.95, $7.95, or $9.95} for albums.

Corporations order from 20-29 line items per order, with a quantity of 1-10 each. Corporations only order albums, at a cost of between $6 and $12 per album.

C.21.3 SDDL Description

```xml
<pool name="itype_ind">
  <choice name="A">
    <weight>30</weight>
    <pool name="prices">
      <choice name="5.95"/>
      <choice name="7.95"/>
      <choice name="9.95"/>
    </pool>
  </choice>
  <choice name="S">
    <weight>60</weight>
    <pool name="prices">
      <choice name="0.95"/>
    </pool>
  </choice>
  <choice name="V">
    <weight>10</weight>
    <pool name="prices">
      <choice name="19.95"/>
      <choice name="24.95"/>
      <choice name="29.95"/>
    </pool>
  </choice>
</pool>

<variable name="orderDay" type="int">
  <iteration base="0" count="#2007-01-01#-#1990-01-01#"/>
</variable>
<variable name="cpool" type="int">
  <queryPool query="select customer_id from Customer order by customer_id"/>
</variable>
<variable name="oidx" type="int">
  <iteration base="1" count="5+(orderDay/150)+IRND(10)"">
    <itervar name="cid" type="int" init="cpool"/>
    <itervar name="isIndividual" type="bool" init="cpool % 2 == 0"/>
  </iteration>
</variable>
```

329
This table is constructed by iterating through each day from Hallux inception (January 1, 1990) through December 31, 2006. For each day, a certain number of orders is specified, which depends upon the date; earlier dates get less orders, and later days get
more orders. For each order, a random customer is chosen, and a count of line numbers is calculated – for individuals it is 1-6, for corporations it is 20-29. For each day, a possible pool of albums is constructed by choosing all albums that were released on or before that day; once an album is chosen, a pool of songs from that album is generated.

The temp_custid field, which is not specified in the schema, has been added. It stores the customer_id associated with each order, and is used later during the construction of the Order_Header pool.

C.22 Order_Header

The Order_Header table contains an entry for every order ever received by Hallux Productions.

C.22.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order_ID</td>
<td>Unique id for the order (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Order_Date</td>
<td>The date and time the order was placed</td>
<td>DATETIME</td>
</tr>
<tr>
<td>Promise_Date</td>
<td>The promised date of delivery, if needed</td>
<td>DATE</td>
</tr>
<tr>
<td>Order_Source_ID</td>
<td>Identifies the source of this order (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Subtotal_Amount</td>
<td>Total amount of the order before tax</td>
<td>DECIMAL(18,2)</td>
</tr>
<tr>
<td>Tax_Amount</td>
<td>Amount of tax to be added to the order</td>
<td>DECIMAL(18,2)</td>
</tr>
<tr>
<td>Customer_ID</td>
<td>Identifies the customer placing this order (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Total_Amount</td>
<td>Total amount of the order with tax added</td>
<td>DECIMAL(18,2)</td>
</tr>
</tbody>
</table>

C.22.2 Rules

The Order_Header table reflects aggregated information derived from the Order_Detail table. On top of the subtotals gleaned for each order from the
Order_Detail table, the Order_Header table applies tax and comes up with a new total. The tax rate will be between 5% and 10%. Individuals always order through “T-Jams” (Order_Source_ID = 4); corporations can order through any order source.

C.22.3 SDDL Description

```xml
<constant name="secs_per_day" type="int" value="60*60*24"/>
<table name="Order_Header">
    <field name="Order_Id" type="int">
        <iteration query="select Order_Id, sum(Quantity*Unit_Price)
        subtotal, min(temp_custid) custid from Order_Detail
        group by Order_Id
        order by Order_Id"/>
    </field>
    <variable name="oday" type="int">
        <formula>(Order_Id-1000000)/100</formula>
    </variable>
    <field name="Order_Date" type="timestamp">
        <formula>#1990-01-01 00:00:00# +
        (oday*secs_per_day)+
        IRND(secs_per_day)</formula>
    </field>
    <field name="Promise_Date" type="date">
        <formula>#1990-01-01#+(oday+5+IRND(5))</formula>
    </field>
    <variable name="sourcePool" type="int">
        <queryPool query="select order_source_id from Order_Source
        order by order_source_id"/>
    </variable>
    <variable name="isIndividual" type="bool">
        <formula>(0+Order_Id$custid)%2==0</formula>
    </variable>
    <field name="Order_Source_Id" type="int">
        <formula>isIndividual?4:sourcePool</formula>
    </field>
    <field name="Subtotal_Amount" type="DECIMAL(18,2)">
        <formula>0.0+Order_Id$subtotal</formula>
    </field>
    <variable name="tax_rate" type="DECIMAL(3,2)">
        <formula>(5+IRND(6))/100.0</formula>
    </variable>
    <field name="Tax_Amount" type="DECIMAL(18,2)">
        <formula>tax_rate*Subtotal_Amount</formula>
    </field>
    <field name="Customer_Id" type="int">
        <formula>0+Order_Id$custid</formula>
    </field>
    <field name="Total_Amount" type="DECIMAL(18,2)">
        <formula>Subtotal_Amount+Tax_Amount</formula>
    </field>
</table>
```
There is a small amount of extra work when computing “Order_Date”, because it is a timestamp (date + time) derived from a simple date. It is simply assigned a random time during the order date.

C.23 Performance

The Performance table tracks all performances by all Hallux bands.

C.23.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance_ID</td>
<td>Unique id for the performance (PK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Performance_Date</td>
<td>The date of the performance</td>
<td>DATE</td>
</tr>
<tr>
<td>Venue_ID</td>
<td>The id of the venue for this performance (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Revenue</td>
<td>The amount made at this performance</td>
<td>DECIMAL(18,2)</td>
</tr>
<tr>
<td>Band_ID</td>
<td>The band playing for this performance (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Agent_ID</td>
<td>The agent for this performance (FK)</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

C.23.2 Rules

Each band gives 10-40 performances after each album it releases. Performances should be about a week apart, on average. Revenue for a performance should have a 50% chance of being between $1000-$5000, a 40% chance of being between $5000-$10000, and a 10% chance of being between $10000-$100000.

C.23.3 SDDL Description

```xml
<table name="Performance">
  <field name="Band_Id" type="int">
    <iteration query="select band_id, album_id, release_date
```
C.24 Customer_Profile

The Customer_Profile table contains information about each individual customer; it does not contain information about corporate customers.

C.24.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profile_ID</td>
<td>A Customer_ID that identifies the customer for this profile (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Favorite_Band</td>
<td>A Band_ID that represents the customer's favorite band in our database (FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Favorite_Album</td>
<td>An Album_ID that represents the customer's favorite album in our database (FK)</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
### C.24.2 Rules

There should be a `Customer_Profile` entry for each individual (non-corporate) entry in the `Customer` table. The `Favorite_Band`, `Favorite_Album` and `Favorite_Song` fields should be accurate (i.e., derived from actual order counts) 80% of the time, and random 20% of the time.

### C.24.3 SDDL Description

```xml
<table name="Customer_Profile">
  <field name="Profile_ID" type="int">
    <iteration query="select customer_id,gender from Customer
                   where customer_id mod 2 = 0
                   order by customer_id"/>
  </field>
  <!-- Random album pool. -->
  <variable name="RAlbum" type="int">
    <queryPool query="select album_id from Album
                   order by album_id"/>
  </variable>
  <!-- Random song pool. -->
  <variable name="RSong" type="int">
    <queryPool query="select song_id from Song
                   order by song_id"/>
  </variable>
  <!-- Random band pool. -->
  <variable name="RBand" type="int">
    <queryPool query="select band_id from Band
                   order by band_id"/>
  </variable>
  <!-- Most-purchased album by this customer. -->
  <variable name="FAlbum" type="int">
    <queryPool query="select item_id, count(*) cnt
                   from Order_Detail
                   where item_id &lt; 50000 and
                   temp_custid = [Profile_ID]
                   group by item_id
                   order by cnt desc limit 1"/>
  </variable>
  <!-- Most-purchased song by this customer. -->
</table>
```
Note that some knowledge of Album_Id numbering (all less than 50000) and Song_Id numbering (all greater than or equal to 400000) were necessary to properly formulate the FAlbum and FSong queries. Also, a short cut was taken for
Favorite_Band: rather than meticulously counting how many albums/songs an individual had ordered from each band, Favorite_Band was defined as “Band associated with favorite album”.

C.25 Customer_Genre

The Customer_Genre table contains genre preference information about each individual customer.

C.25.1 Schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genre_ID</td>
<td>The id of a genre that is one of the customer's favorite genres (PK,FK)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Customer_ID</td>
<td>The id of the customer that has the genre as a favorite (PK,FK)</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

C.25.2 Rules

This table should be derived from recorded ordering patterns exhibited by customers.

C.25.3 SDDL Description

```xml
<table name="Customer_Genre">
  <field name="Profile_ID" type="int">
    <iteration query="select profile_id, favorite_band, favorite_album, favorite_song from Customer_Profile order by profile_id"/>
  </field>
  <variable name="FavBand" type="int">
    <formula>0+Profile_ID$favorite_band</formula>
  </variable>
  <variable name="FavAlbumBand" type="int">
    <formula>(0+Profile_ID$favorite_album)/10</formula>
  </variable>
  <variable name="FavSongBand" type="int">
    <formula>(0+Profile_ID$favorite_song)/200</formula>
  </variable>
</table>
```
For each customer, the Genre_ID values assigned are “Genre(s) associated with favorite band, genre(s) associated with favorite album’s band, and genre(s) associated with favorite song’s band.” The favorite band/album/song information is taken directly from the previously generated Customer_Profile table.