Le Generator: A Test
Data Synthesis Framework

J A K O B  B E R G E N D A H L

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2009
Le Generator: A Test
Data Synthesis Framework

J A K O B B E R G E N D A H L

Master’s Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2009
Supervisor at CSC was Stefan Arnborg
Examiner was Stefan Arnborg

TRITA-CSC-E 2009:117
ISRN-KTH/CSC/E--09/117--SE
ISSN-1653-5715
Le Generator: A Test Data Synthesis Framework

Abstract

Test data is needed to test any database application, but of extra importance when testing business intelligence/data warehousing systems, since they often require a larger volume of data and also often lack a user interface for entering data.

This thesis describes Le Generator—a prototype of a framework for test data synthesis implemented in the Python programming language. I describe background, objectives and implementation. I also experiment with the prototype and evaluate its usability, versatility and performance.

I finish with some analysis of the results, a few recommendations for similar future endeavors and a few suggestions for improvement, such as having clear priorities regarding usability/versatility, choosing a declarative and not Turing complete metadata specification language and making the user interface discoverable.

Le Generator: Ett ramverk för syntes av testdata

Sammanfattning

Test data behövs för att testa alla databasapplikationer, men är extra viktigt för att testa business intelligence-system/datalager, då dessa ofta kräver större datavolymer och även ofta saknar användargränssnitt för datainmatning.


Jag avslutar med analys av resultaten, några rekommendationer för framtida liknande projekt och några förbättringsförlag, såsom att ha tydliga och konsekventa prioriteringar angående användbarhet respektive mångsidighet, välja ett deklarativt och ej Turing-fullständigt språk för metadataspecifikation och att ha ett upptäckbart användargränssnitt.
Preface

Although this is a work of my own, I owe thanks to a lot of people for help and support in the process of writing it. This is the section where I mention some of them, in no particular order. I would like to direct my very special thanks to:

- My parents Peter and Kristin, for their seemingly infinite support and encouragement. Also for pushing me to actually finish writing.
- My supervisors and co-workers at LucidEra, for helping me throughout the project with lots of things, including but in no way limited to: acquiring a US visa, teaching me their build environment, explaining concepts such as star-schemas, OLAP cubes and bitmap indices, helping me establish requirements and scope for the project, testing the prototype as well as commenting on the data quality and usability.
- Professor Stefan Arnborg, my advisor at CSC, KTH, for comments and guidance throughout the project.

I apologize in advance to anyone who deserved my thanks but who did not make the list. With that taken care of, let us get to the actual report!
# Table of Contents

1. Introduction ............................................................................................................. 1
2. Background .............................................................................................................. 2
   2.1 Defining Business Intelligence ........................................................................ 2
   2.2 Software Testing ............................................................................................... 3
3. Problem Definition .................................................................................................. 4
   3.1 Aim .................................................................................................................... 5
   3.2 Scope .................................................................................................................. 6
   3.3 Specifying Metadata ......................................................................................... 6
   3.4 Dependency Modeling ...................................................................................... 6
      3.4.1 Resolving Dependencies .......................................................................... 7
   3.5 Intermediate Storage ....................................................................................... 7
   3.6 Data Types ......................................................................................................... 7
4. Related Work ........................................................................................................... 9
5. Method ..................................................................................................................... 10
   5.1 Metadata Specification Language Survey ....................................................... 10
      5.1.1 SQL ........................................................................................................... 10
      5.1.2 Python ..................................................................................................... 10
      5.1.3 Java ........................................................................................................ 11
      5.1.4 XML and other mark up languages ......................................................... 11
      5.1.5 Custom Language .................................................................................... 11
   5.2 Le Generator Overview ..................................................................................... 11
      5.2.1 First Implementation .............................................................................. 11
      5.2.2 Second Implementation .......................................................................... 11
   5.3 Data Dependencies and Relationships ............................................................ 12
      5.3.1 Specification ........................................................................................... 12
      5.3.2 Resolving Inter-Table Dependencies ....................................................... 13
   5.4 Synthesizing Primitive Data Types ................................................................. 14
      5.4.1 Words ....................................................................................................... 15
      5.4.2 Texts ......................................................................................................... 15
      5.4.3 Persons ..................................................................................................... 16
      5.4.4 Companies .............................................................................................. 16
      5.4.5 US Addresses ......................................................................................... 16
      5.4.6 Seed Data ............................................................................................... 16
   5.5 Statistical Analysis of Customer Data ............................................................... 16
6. Experiments ............................................................................................................ 18
1 Introduction

This thesis is based on a project performed at LucidEra, Inc. in San Mateo, California, USA from February through July in 2008.

Chapter 2: Background provides a brief overview of the business intelligence, data warehousing and software testing. I ask the question “Why do we need to generate test data?” and answers with a few reasons. I also introduce some common words and phrases used in the rest of the thesis.

In chapter 3 I describe the aim and scope of the project. I also identify a few problem areas that need to be developed and investigated. Those areas are metadata specification, dependency modeling and primitive data modeling.

Chapter 4 outlines some previous research into the subject of test data synthesis.

Chapter 5: Method describes the iterative process used to implement a prototype system, called “Le Generator”, attempting to solve the problems described in chapter 3.

In chapter 6 I test my prototype by generating some data. I perform a qualitative analysis of the ease of use and versatility of the framework and a quantitative analysis of the performance and scalability. I generate some natural language fragments based on data in the CRM system of a real-world, but anonymous, LucidEra customer using a few different methods.

In chapter 7 I present qualitative and quantitative findings of the experiments outlined in chapter 6.

Chapter 8 contains discussion, lessons learned and future outlook. I give a few recommendations for future projects in this area.

I finish with the bibliography in chapter 9.
2 Background

This chapter gives a brief overview of business intelligence, data warehousing as well as software testing in general and introduces some common vocabulary used in the rest of the thesis.

Business Intelligence, or BI, is a rapidly growing industry. Studies by the research company IDC (Vesset & McDonough, 2008) (Vesset & McDonough, 2007) indicate a global revenue growth for the BI tool sector of over 11% annually between 2004 and 2007, reaching 7.05 billion US dollars, with no signs of stopping. Since acquiring the tools constitutes only a fraction of the total cost of a typical BI project, the total global spending on BI is likely to be much greater. Let us take a closer look at the technology used in this growing market.

2.1 Defining Business Intelligence

Business intelligence is the process of offering actionable information to decision makers about their business. The term most commonly refers to data based decision support systems, where a number of user facing BI applications allow users to access an underlying data warehouse.

There are two main schools of thought in the data warehousing world: That of Ralph Kimball and that of Bill Inmon. I will only use the Kimball model as described in (Kimball, Ross, Thornthwaite, Mundy, & Becker, 2007) which is the one used at LucidEra.

A data warehouse in the Kimball model typically consists of two parts: An Extract, Transform and Load (ETL) system, which pulls data from various source systems and transforms it for business consumption, and a Business Process Dimensional Model, a model that is optimized for both business user ease of use and query performance.

The ETL system is responsible for data extraction from source systems, error and reasonable data quality checking, data cleansing (by for instance removing duplicates or consolidating slightly different names for the same entity), and finally data transformation into the warehouse schema. This can be done in several stages: The first stage loads modified or inserted data from the source systems into an Operational Data Store, ODS, which uses the source schema. The second stage transforms the data into a source independent staging schema. Stage three transforms data from the stage tables into the final warehouse schema. Finally, the post-load processes are run. The schemas and processes just described are often referred to as the ETL pipeline, with data flowing through it.

Dimensional models, when realized as star schemas in a relational database, consist of fact tables and dimension tables. A fact table contains metrics resulting from an event (such as a closed deal) and foreign keys to its associated dimensions (such as date, product, customer or sales rep). A diagram of such a schema looks like a star, hence the name.

A fact table and its related dimensions form a multidimensional cube. A multidimensional query defines an aggregate function, dimensions for each axis and often filters on other dimensions, returning a slice of the cube. An example of a multidimensional query is “closed opportunity amount in the North American region by sales rep and fiscal quarter”. Here the metric is “opportunity amount”, from the “opportunities” table using “sum” as the aggregate, the axis dimensions are “sales rep” and “date” and it filters on the opportunity status and geographical dimensions to only include closed opportunities in North America.

A database system optimized for quickly answering multidimensional queries is often referred to as an online analytical processing, or OLAP, database management system (as opposed to more common online transactional processing, OLTP, database management systems like Oracle, Microsoft SQL Server or MySQL). OLAP-databases implemented as described above, with star schemas in relational databases, are sometimes referred to as relational OLAP, or
**ROLAP.** The underlying relational database can be a traditional OLTP database or a relational database or it could be a *column store database*, which stores data by column instead of by row to increase query performance at the cost of insert and update performance.

Most business users will not formulate queries against the warehouse themselves. BI applications, ranging in complexity from static reports to dynamic online analytical applications, enables users to access the warehouse in quickly and easily.

### 2.2 Software Testing

In *Effective Methods for Software Testing* (Perry, 1995), William Perry defines testing to include three concepts:

- The demonstration of the validity of the software at each stage in the software development lifecycle.
- Determination of the validity of the final system with respect to user needs and requirements.
- Examination of the behavior of a system by executing the system on sample test data.

He goes on to cite a U.S. General Accounting Office study, “Improvements Needed in Managing Automated Decision-making by Computer throughout the Federal Government” (FGMSD-76-5), which states that input data is frequently a problem: Incomplete, incorrect and obsolete data were all common problems. Perry concludes that these problems should be included in any test program.

In Part 1 of the book, titled “Developing a Test Approach”, Perry describes risk analysis as a way to establish a testing policy. Based on that policy, Perry describes how to develop a testing strategy and based on strategy how to choose testing tactics.

Perry defines some common testing concepts to be aware of when designing your testing tactics:

- “White box testing” is when the tester is familiar with the inner workings of the artifact under test and verifies not only input and output, but also the internal logic and state transitions.
- “Black box testing” is when the tester is not aware of the internal mechanisms of a system. The tester can only feed input to the system and verify that the output is correct.
- “Regression testing” is testing each new version of an artifact to make sure old faults do not re-appear. A test data generator is especially useful for this: Whenever you find a bug triggered by a certain data pattern, update the test data scripts to make sure that data pattern is included in a test case in the regression suite.

The cost of defects increases the later in the software development lifecycle they are detected. Perry proposes a lifecycle testing approach, testing deliverables during each stage of the development - requirements, design, programming, test, integration and maintenance.

For the test phase, Perry recommends the use of a “test data test tool” and a “volume test test tool”. Both these tools are concerned with generating correct data, incorrect data and test data exceeding internal limits. Perry acknowledges that “few organizations allocate sufficient budgets for this type of testing” and that “…personnel are not trained in the use of these test tools”.

In an appendix, Perry cites a survey by the Quality Assurance Institute conducted at their Software Testing Conference in 1994. Eighty attendees responded. Respondents planned 45% on average of the software development schedule to testing, but ended up using only 27%.

Despite this, the testing budget was often 23% of the project budget but ended up using 24% of the project budget. So even though the time used was reduced by 40% compared to the time scheduled, the expenses increased by over 4% compared to budget. This means that the burn rate during the software-testing phase was 74% higher than planned—a huge increase.
3 Problem Definition

This chapter describes the project I set out to do, including the aim and scope. Later I break it down into a few problem areas and describe some common vocabulary used in the rest of the thesis.

The LucidEra Platform is a set of generic tools for creating metadata driven business intelligence applications. It includes a data warehouse running the free software and open source LucidDB database engine¹, a tool for interacting with the metadata repository and creating the BI applications called “the workbench”, a web based configuration and management interface called “the Configurator”, as well as a web based report builder and viewer called ClearView.

To produce a report, ClearView issues multidimensional expression (MDX) queries to a Mondrian² relational online analytical processing (ROLAP) engine, which in turn issues structured query language (SQL) queries to the LucidDB warehouse. LucidDB uses plug-ins called connectors to access source systems. These source systems are made visible to applications using the Management of External Data extension to the SQL standard (SQL/MED), abstracting the data origin so that remote services can be queried just like local tables.

Data gets into the system via an ETL process, which pulls data from source systems like SalesForce.com, Oracle, NetSuite or Microsoft CRM into the warehouse. The ETL process consists of several stages, with separate storage for each stage: operational data store, staging tables and warehouse tables (see Defining Business Intelligence above).

1. Data is extracted from source systems and put into an operational data store (ODS), a copy of the source database using the source schema.
2. Data is transformed by a business adaptor, a piece of software specific to that source, into a staging schema common to all sources and loaded into the staging tables.
3. Data is transformed into an online analytical processing (OLAP) oriented star-schema and loaded into the data warehouse. Post load processing takes place.
4. In order to analyze time-dependent trends, the LucidEra analysis platform can periodically save snapshots of the source data. This is often the only way to add a time dimension to the analysis, recording changes which are otherwise not kept in the source data.
5. In order to test this entire set of tools, data is needed. There are a variety of source systems supported, each using a different schema with many tables each. Obviously, creating test data by hand is both error-prone and tedious. Automation is needed.

The problem LucidEra was facing was supply of test data. You need data before you can test any database system. Of course, for many simple transactional applications testers can simply create their own ad hoc test data from within the application itself as part of their testing. For BI applications however, the overhead of creating ad hoc data is greater than for most other applications, for two reasons: First of all, BI applications like those of LucidEra often lack a user interface for adding data to the system. These systems are designed to extract data from other source systems, not from the users. Another important reason is that the volume of data required to test common BI functionality, such as data aggregation and slicing, is greater than for most transactional systems, where decent coverage can be obtained with only a few records per table.

¹ http://www.luciddb.org/
² http://mondrian.pentaho.org/
Problem Definition

Their current approach was to extract data from the source systems of an actual customer and “scramble” it, altering data in an attempt to make it impossible to identify the source and, in case of identification, obscuring sensitive information enough to make it useless.

This approach had the following problems:

- Using sensitive customer data for testing exposed LucidEra to legal liability. Even scrambled data leaks information. For instance the number of records in each table translates directly to key performance metrics such as the number of leads, opportunities, orders and accounts. When deciding a scrambling strategy a trade-off between the privacy of the source on the one hand and semantically meaningful data on the other hand must be made.
- The source data was often subject to constraints and dependencies, which were difficult not to violate during scrambling. One way to keep at least the foreign key relationships intact is to use a one-to-one function for scrambling each value.
- The data set size was fixed. If more or less data is needed, existing data had to be truncated or multiplied in a non-obvious way.
- The output was unpredictable. Since data could not be controlled, each time new data was used, correct results of tests had to be calculated by hand or by using a version of the system under test assumed to be correct (in effect reducing tests using the data to a regression test).
- The data could not easily be used to test snapshot functionality. In order to test the periodic snapshot functionality of the platform periodic snapshots needed to be collected from a source and then scrambled. This could take days or even weeks depending on how many snapshots were required.

3.1 Aim

The goal of this project was to investigate the needs for different kinds of synthetic data and to provide an extensible framework that could fulfill as many of those needs as possible.

After a meeting with Boris Chen, my project advisor and director of engineering at LucidEra, I established a feature wish list for the data synthesis tool. It included the following features:

- The size of the generated data sets must be configurable.
- Different field types must be supported, with different data domains and distributions. The user must be able to add new field types.
- The generated data must respect constraints in the data model. The user must be able to specify constraints and data dependencies.
- It must be possible to describe expected output using constraints on the data (e.g. sum of opportunity amount = some value which can be automatically verified after ETL).
- The tool must provide a language for specifying data-types, constraints and other properties.
- The tool must also be able to simulate changes over time and output periodic snapshots.
- The tool should be able to generate non-conforming data for testing of error handling.
- The tool should be able to extract meta-data from the platform to create templates for data models.
- The system should be able to perform statistical analysis of production data in order to generate synthetic data with statistical properties from production data.
- The tool should be able to mix data from multiple sources, such as hand-crafted data, scrambled production data as well as previously generated data.
- The tool should be able to produce pronounceable random text for fields, using some statistical model.
3.2 Scope

We also agreed on a few limitations to the scope of the project. These were:

- The system will not feature any graphical user interface. The user will interface with the system using the command line. This will also simplify automation through the use of shell scripts, make-files, etc.
- The system is only required to run on Ubuntu Linux and Red Hat Enterprise Linux 5 systems on x86 processors.
- Distributed synthesis will not be directly supported by the tool. Performance is not an important goal for the first version.
- Input and output datasets will be in plain-text (with comma-separated values) format.

3.3 Specifying Metadata

The first task when generating data is gathering metadata. This is a rather vague term, defined by Merriam-Webster Online Dictionary simply as “data that provides information about other data”. In this thesis, metadata includes both technical information such as the data-type of the underlying database field, the field length in case of variable length fields, the acceptable values as well as semantic information: what real-world piece of information is contained in the field, is it an address, somebody's first name, an email address or something else?

One source of metadata is the database schema, if available. For modern database applications, the schema typically includes information about data types, field lengths, UNIQUE-constraints, primary keys and foreign keys. A source of metadata that is often overlooked is the table and field names, which in well designed database applications convey hints of the semantic contents of the fields.

It quickly became clear that the user would have to supplement the existing metadata with some extra information in order to get meaningful synthetic data. Clearly, a large part of the problem was deciding what language to use for metadata-specification. In chapter 5.1, Metadata Specification Language Survey, I present a survey of different language choices considered.

This definition of metadata also includes information about dependencies in the data, ranging from formally specified like foreign key relationships, to semantic like the way first names depend on genders, to subtle like how first names also depend on the parents’ income, as shown in (Levitt & Dubner, 2006).

3.4 Dependency Modeling

When a piece of data in a schema depends on other data, it needs to be synthesized with this dependency in mind. Houkjær, et. al. (Houkjær, Torp, & Wind, 2006) identified four different types of data dependencies: foreign keys, intra-row, intra-column and inter-table.

A foreign key is a set of fields in a table that take the value of one of the primary keys of table, thus uniquely identifying a row of that table. It is a special case of an inter-table dependency. For each foreign key relationship the cardinality of a primary key in that relationship is the number of foreign keys referencing that primary key. Since this property is unique for each key value it cannot be specified directly when generating random keys. It can however be modeled statistically for the entire relationship.

Each foreign key relationship also has the following properties: The participation of a relationship is the fraction of primary keys referenced by any foreign keys. If NULL is

3 http://mw4.m-w.com/dictionary/metadata
permitted for the foreign key fields, the \textit{NULL-fraction} is the fraction of foreign keys that are NULL.

An intra-row dependency occurs within the same row. For instance, in a table where each record corresponds to a person, there would be a strong correlation between the person’s gender and first name. Reasonably realistic models for data synthesis would have to include knowledge of one of these while generating the other. More advanced models could also include the person’s age, since popularity of different names changes over time. Another example would be records with duration, like start/end or created/modified, where one date needs to be earlier than the other (because you need to start an activity before finishing it, and create a record before modifying it).

An intra-column relationship is when data in a column depends on other values in the same column. Examples include time-series with increasing timestamps and slowly varying measurements.

An inter-table dependency exists when data in one table somehow depends on data in another table. In the most general case, data could depend on data in any number of records in several other tables.

Hierarchies are a special type of dependency, usually modeled as a foreign key from a table to itself. This structure allows any directed graph to be represented in the general case, but in many cases the application expects a rooted tree with a relatively small depth.

\subsection*{3.4.1 Resolving Dependencies}

Once dependencies have been identified, they need to be resolved during data generation. Dependent fields need to be generated after the fields they depend on. In chapter 5.3.2, Resolving Inter-Table Dependencies, I describe a method to generate data while resolving dependencies.

\section*{3.5 Intermediate Storage}

In order to resolve dependencies, a data synthesizer clearly needs access to the data while it is being generated. The simplest way would be to simply stream the data to disk as it is being generated and read it back in for access. For anything but trivial schemas, this would be horribly inefficient, requiring disk access for even the simplest dependency.

I considered two options for intermediate storage, in-memory representations and databases. In memory storage has the advantage of being very fast and easy to implement for simple cases like intra-row dependencies. For more interesting cases however, implementation quickly becomes a daunting task and unless great care is taken, both CPU usage and memory use would sky-rocket.

A database engine has the advantage of being capable of using indexed look-ups to speed up queries, and comfortably handles any dependency that can be expressed in a set of SQL-queries. However, it increases code complexity and introduces performance overhead for the very simple cases.

\section*{3.6 Data Types}

To generate realistic data, we need to identify the primitive data types that make up a record. In order to make the system extensible it makes sense to factor out the routines for generating common data types into separate modules.

Common string types include names, words, sentences, addresses, company names, internet domains and phone numbers. This data can be made more realistic by using seed data such as dictionaries, lists of common names or a table mapping zip codes to cities.
Independent numbers can be sampled from statistical distributions such as the normal, binomial, Zipf (a special case of the Zeta distribution), uniform and Poisson distributions.
4 Related Work

As part of my research I tried to find out what others had done in the field of test data generation. This chapter summarizes papers I found relevant.

The authors of (Chays, Dan, Frankl, Vokolos, & Weber, 2000) describe a system that can take distinct data values (reasonable choices) specified by the user and put them into database tables. Their system can also parse SQL to find foreign keys as well as UNIQUE-constraints. It does not deal with random values however, so test are restricted to values specified by the user.

In Flexible database generators (Bruno & Chaudhuri, 2005), the authors describe a language called DGL (data generation language), used to control the generation of data. Its data types are scalars, rows and iterators. Iterators create rows, which consist of scalars. It can sample a set of standard random distributions such as normal, uniform, exponential, Zipfian and Poisson. Dependencies in data must be explicitly handled by the user using two functions called persist and query. The order the tables are synthesized in is also left to the user to decide.

The system described in (Stephens & Poess, 2004) aims specifically to generate test data for business systems. Its main feature is performance and scalability. The authors claim that it is capable of generating 100 Terabyte of data in hours, using multiple processors or clusters. The paper also covers some common BI concepts such as hierarchies and slowly changing dimensions.

(Wu, Wang, & Zheng, 2003) extract statistical rules from a production database and use those rules to synthesize data that is “close-looking”. They want data that has the same performance characteristics as the production data when used in the application under test.

QAGen, described in (Binnig, Kossmann, Lo, & Özsu, 2007), includes queries tested as input. This allows the system to create data tailored to each specific query. It can also constrain desired output values such as the cardinality of a certain JOIN. This feature facilitates end to end testing of a query execution engine. However it is not very useful for synthesis of general purpose test data for ad hoc testing.

A slightly older paper also mainly concerned with scalability on multi-processor systems is (Gray, Sundaresan, Englert, Baclawski, & Weinberger, 1994). It mostly discusses fairly low-level implementation details for parallel database generators, such as process spawning and table partitioning strategies. It also covers sampling of different random distributions, using techniques described by Donald Knuth in volume 2 of The Art of Computer Programming.

(Houkjær, Torp, & Wind, 2006) describe a system for data generation similar to the one I implemented. They introduce the data dependency graph and describe a modified topological sort, capable of breaking cycles by modifying the graph, used to determine the synthesis order.
5 Method

This chapter describes the development of my prototype tool “Le Generator”, how I decided on the metadata language issue, how dependencies are modeled, how plug-ins work, how data dependencies are handled as well as how the tool can be used to fit cardinalities in generated data to distributions found in input data.

The schema for the Universal Flat File Adapter of LucidDB comes in a number of BCP-files, one per table. BCP is a format invented by Microsoft to control the bulk copy utility of SQL Server. It contains the field names, types and lengths, as well as information about how to parse the data files. No inter-table relationships or other constraints are included, however. In order to get a hold of those, I would have to query the metadata-repository directly.

5.1 Metadata Specification Language Survey

Choosing a method of metadata specification is a trade-off between power and usability. In one end of the spectrum the user has complete control of the data generated, and in effect writes a custom generation tool every time. This approach puts all control in the hands of the user. However, the detailed instructions required makes writing data specifications tedious and prone to error. Adding some restrictions, the framework could handle all the heavy lifting: input, output, schema parsing, and data-dependencies, while still providing a Turing complete language for the user. Other tools have restricted the user even more, only allowing the user to choose data-type for each field from a list. A hybrid approach could allow combining primitive types with expressions for ad hoc extension of the basics. On the other end of the spectrum the tool is fully automatic, requiring no further input than the existing metadata.

The data generation tool was only to be used by software engineers, so I could safely assume that the end user would be familiar with at least one programming language.

5.1.1 SQL

As with almost any application involving relational data, SQL is an option. In this case, one could augment the data definition part of SQL with extra data-types and dependencies. Another approach would be to simply write standard SQL INSERT statements, calling user-defined functions that returned values with the desired properties. In order to benefit from this approach, one would have to integrate the solution into an existing database engine. In terms of expressiveness and power, standard SQL is notably not Turing complete, however most practical implementations are.

Anyone familiar with relational databases would be immediately familiar with SQL, but would still have to learn the data synthesis extensions.

5.1.2 Python

Python is a dynamic object-oriented programming language that has gained a lot of attention lately and been adopted by large companies such as Google (Hamilton, 2008). The main Python implementation is open source and free even for commercial use.

Python can be used to specify metadata in at least two ways: Either the user specifies python expressions for each field, or the user writes python modules. In both cases special library functions are called to specify additional metadata such as dependencies or record counts.

Python is already the language of choice for much of the supporting scripts used at LucidEra.

---

5.1.3 Java

Java is a statically typed object-oriented programming language used for most of LucidEra's existing software. To specify metadata in Java, the user would write and compile classes that would be loaded dynamically into the data generation framework. The advantages of Java are performance and familiarity. Every engineer on the LucidEra platform team knows Java. The performance of a typical Java implementation greatly surpasses that of Python (Nene, 2008) (Cowell-Shah, 2004).

However, Java is more verbose than Python, requiring more code to get started (at least a class and method declaration), lacking features such as list comprehension and requires blocks to be explicitly enclosed with curly braces. The overhead of specifying classes would be a significant part of the required code. A separate compile step is also needed, which would make updating metadata more tedious.

5.1.4 XML and other mark up languages

By using a dialect of XML (eXtensible Markup Language) one could potentially gain all the advantages of a custom language while leveraging the entire infrastructure built around XML. That would mean user-defined semantics but almost free parsing and validation. The same can be said for other markup languages such as YAML (Yaml Ain't Markup Language, a human friendly data serialization standard) and JSON (JavaScript Object Notation, a lightweight data interchange format).

While a mark-up language dialect can be Turing complete, such a dialect is almost always more verbose than Python or even Java when expressing short programs. For a more declarative approach, one of these mark-up languages would be a very good fit.

5.1.5 Custom Language

Designing a custom language for metadata specification could potentially yield the best fit to the problem domain, but at the cost of significant implementation overhead. I chose not to, simply because designing a language that is better than all of the above is a pretty daunting task in itself. Also, by using an existing language I would not have to write my own lexing and parsing.

5.2 Le Generator Overview

I tried implementing two different approaches, both using Python.

5.2.1 First Implementation

The first implementation required one Python expression per table in the dataset. That expression was evaluated once for each record to be generated and was supposed to return a “dict”-object (the Python version of what is also known as a map, dictionary or associative array). That implementation ran into a number of limitations when trying to specify dependencies in the data.

Since it worked by executing the Python expressions that evaluated to dictionaries (a built in type in Python that maps keys to values), there was no way for them to have intra-row dependencies. Values for one key of the dictionary could not depend on values for other keys since the other keys have not been stored when the dictionary is evaluated.

5.2.2 Second Implementation

I attempted to improve on the first implementation by allowing entire Python modules to be written for each table. In order to simplify writing data templates, these modules are executed in
the same namespace as a set of generators (plug-ins that generate primitive or compound data
types), so no import statements are normally needed in these modules.

These modules are executed at least once per record to be generated (see the Data Dependencies
and Relationships chapter below). Every time a module is executed, its local variables are reset
to their original values.

Specifying a field is as easy as declaring a local variable. All local variables that match field
names in the database schema are automatically saved in the resulting dataset.

Some names have special meaning within the context of metadata template.

One of these reserved names is Table, which refers to an object which is saved between
different records but unique to each table in the schema. This object controls certain properties
of the current table and can be used to store data between runs.

One important property of the table object is number_of_records. This variable could be set to
control how many records to generate for this specific table, or unset to let the framework
compute the number of records needed.

In order to get started quickly with the metadata specification, the tool can generate a scaffold, a
set of empty metadata templates, given a database schema.

5.3 Data Dependencies and Relationships

Once a language had been chosen, the second problem area to address was data dependencies.

5.3.1 Specification

The user handles intra-row dependencies manually. For example, suppose first name was
supposed to depend on gender. Suppose there is a generator function called first_name that
takes an argument gender {‘M’, ‘F’}. Then this dependency could be specified quite elegantly as:

\[
\begin{align*}
gender &= \text{choice}('M', 'F') \\
\text{first\_name} &= \text{first\_name}(gender)
\end{align*}
\]

If this type of dependency is common, it can be refactored into a compound data generator, for
example a class Person, with the constructor accepting the probability of male gender as an
argument. The data template would then look like:

```
p = Person()  # temporary Person instance
gender = p.gender
first_name = p.first_name
```

In case of inter-row dependencies in the same table, data can be maintained in the Table-object.
Say for instance that a table is supposed to contain a time in milliseconds and a measurement
which should not change faster than 0.01 units per 1 millisecond, starting at time 0 and
measurement 0.00. Let us assume that we want between between one and three samples per
millisecond, uniformly. This could be specified as:

```
# Starting values.
if Table.prev_time = None:
    Table.prev_time = 0
    Table.prev_measurement = 0.0

# Calculate current time and measurement.
# U(L,U) is uniform distribution, discrete for int and continuous for float.
Time = Table.prev_time + U(1, 3)
delta_time = Time - Table.prev_time
Measurement = Table.prev_measurement + delta_time * U(-0.01, 0.01)

# Save values for next record.
Table.prev_time = Time
Table.prev_measurement = Measurement
```

This places a restrictions on the dependencies that are possible to express—records can only
depend on records already generated. Of course, in a relational database order is unimportant,
but this tells us that there cannot be two-way dependencies. In the previous example, records containing previous measurements cannot depend on records containing later measurements using this mechanism. Thankfully, such two-way dependencies seem rare in practice.

The last and most general type of dependency is the inter-table dependency. Le Generator supports only a limited subset of these. These dependencies are specified by setting a local variable in the data template to an instance of either of the classes FK or one of its attributes. The FK constructor accepts the foreign table, the cardinality distribution, as well as the participation and NULL-probability. Participation and NULL-probability were defined as fractions of the total number of records in the Dependency Modeling chapter but are interpreted as probabilities for the purpose of data generation. The law of large numbers (Blom, 1989) tells us that for large enough datasets the fractions will converge to the corresponding a priori probabilities. An example to clarify:

Suppose there is a table Users with the fields Id and Name. Also suppose that we want to write a template for the table Accounts, and that each account has an owner (NULL-fraction = 0), half of the users own accounts (participation = 0.5) and the number of accounts per participating user follows a binomial distribution with \( n = 10 \) and \( p = 0.3 \). The schema is denormalized and both the Id and the Name of the owner are stored in the Accounts table. We are only considering consistent data in this example so OwnerName should always equal the Name field of the related User record. The corresponding Accounts data template would look like this:

```plaintext
Owner = FK('Users',
            cardinality=Bin(10,0.3),
            participation=0.5,
            nullfraction=0)
OwnerName = Owner.Name
```

Note that the owner field will take the value of the first field in the foreign table, in this case Id. If the value of the first field is not needed, one could simply specify an attribute of the FK instance as is done on line 2.

This mechanism covers cases where we want to access a value in a single record of a foreign table, including but not limited to actual foreign key relationships. More general cases, such as when we need to join other tables or compute aggregates or subqueries, are not supported. Some additional limitations of this approach will be described later in this thesis.

### 5.3.2 Resolving Inter-Table Dependencies

In the above example, the Owner field of the Accounts table cannot be filled before the Id column of the Users table. Clearly, a data synthesis framework needs to be aware of this, and fill the tables in the correct order.

Each inter-table dependency corresponds to an edge in a dependency graph. Houkjær et.al. (Houkjær, Torp, & Wind, 2006) demonstrated this approach and let the tables in the schema correspond to nodes of the dependency graph. This worked well for non-cyclic dependencies. However, it turns out that cyclic dependencies are not uncommon in real world schemas, and their approach required some tricky graph manipulation to break the cycles.

I chose instead to let the nodes of the dependency graph correspond to \((table, field)\)-pairs. This way the graph is much less likely to contain cycles. In fact, a cycle in this graph would correspond to a field taking a value of a field that changes depending on the value of the first field. The final value of such a field is not guaranteed to be well defined, so cycles in the dependency graph are considered as errors—this means that an error free dependency graph is a directed acyclic graph.

First the data template modules are executed once per table in the schema to discover inter-table dependencies and to construct the dependency graph. The requested number of records is also recorded in this step, if it is set in the data template.
If the number of records is not set, the framework will attempt to compute it using the dependencies. The formula for the approximate number of records in a table with a foreign key on another table with is:

\[ N' = \frac{N \cdot E(C) \cdot p_{\text{participating}}}{1 - p_{\text{NULL}}} \]

Where \( N \) is the number of records in the related table, \( C \) is a random variable describing the number of records for each participating record in the related table, \( p_{\text{participating}} \) is the fraction of participating records in the related table and \( p_{\text{NULL}} \) is the fraction of NULL-records. Note that this is an approximation—the exact number of records needed is not known until the cardinality random variables are sampled.

Next the templates are executed once for each record, to fill in fields that do not participate in any such relationship.

Next the topological sort of the dependency graph is computed (unless there are cyclic dependencies which are detected in this step). A topological sort is a linear ordering of the nodes of a directed acyclic graph such that all nodes reachable from a node \( u \) appear after \( u \) in the ordering. Such an ordering can be efficiently computed and is guaranteed to exist (Cormen, Leiserson, Rivest, & Stein, 2001).

Finally the data templates are executed again in topological sort order to resolve the inter-table dependencies.

Keys are resolved in two phases. In the first phase an availability vector \( A[u] \) is calculated for each record \( u \) in the referenced table. Let \( c_u \) be a value sampled from the cardinality distribution of the relation and let \( p_u \) be 1 with probability \( p_{\text{participating}} \) and 0 with probability \( (1 - p_{\text{participating}}) \). Then:

\[ A[u] = c_u p_u \]

In the next phase the following steps are performed for each record in the table with the foreign key:

1. Set key to NULL and quit with probability \( p_{\text{NULL}} \).
2. Otherwise find a record \( u \) in the referenced table, such that \( A[u] > 0 \).
3. If such a record was found, set key to \( u \) and decrease \( A[u] \) by 1.
4. Otherwise we have too many foreign records, so set key to NULL.

If \( A[u] > 0 \) for any \( u \) after this step, there were too few foreign records. If we reached step 4 in the above algorithm at any point, there were too many (increase a counter in step 4 to find out how many too many). In my application, setting any extra keys to NULL and ignoring any key deficit was acceptable.

Other applications might require this situation to be handled by adjusting the number of records in the tables involved in the relation. Note that this could fail due to multiple constraints leading to an ambiguous number of records.

### 5.4 Synthesizing Primitive Data Types

Synthesizing semantically valid data is an interesting engineering problem on its own. I describe methods to generate a few common types of data. As described above, compound data types can be a powerful abstraction to move some of the tricky intra-row dependencies from the user interface to the data plug-ins.
5.4.1 Words

Excellent sources for words are the dictionaries that come with most operating systems and word processors. One can pick word uniformly from a dictionary with acceptable (and sometimes quite funny) results for many purposes.

For more advanced use, there are word frequency tables for English with word variants and word classes available\(^6\).

5.4.2 Texts

Many databases contain free-text fields for descriptions, messages, etc. Filling these fields can be done in various ways. In my application, the values of these fields were often relatively irrelevant to the function of the system and the most important objectives were readability and distinctiveness.

The simplest strategy was to pick a string of random words and mix with punctuation. This can result in texts that make no grammatical sense, so the readability is somewhat limited.

Another strategy is to allow sentence structure to be specified using a context-free grammar. By randomly picking rules and expanding the text until only terminal symbols remain a grammatically valid text can be generated. Allowing special terminal symbols such as verb, name, or other generator functions makes this approach very flexible. Rules can also be given probabilities to further increase the naturalness. This approach produces text with high readability but requires increasingly complex grammars to produce a large number of distinctive texts.

Another strategy that I implemented was based on Markov chains of words and an example of a Markov chain Monte Carlo (MCMC) method. An \(n\)-order Markov chain is a set of states \(S\) and a set of transition probabilities \(P: S^n \rightarrow S\), where \(P(s_t, s_{t-1}, ..., s_{t-n})\) is the probability of transitioning to the state \(s_t\) given the \(n\) previous states \(s_{t-1}, s_{t-2}, ..., s_{t-n}\).

For the purpose of generating natural language, the state set consists of words and special symbols for the start and end of the text. One nice property of Markov chains is that they can be constructed from existing texts. In my case I used the texts in an actual production database.

An \(n\)-order Markov chain can be represented as a graph with weighted edges, where the nodes are sequences of \(n\) states and the edge weights are the transition probabilities. To construct such a graph from a set of texts:

1. For each text:
   a) Start at the node consisting of only the special start symbol.
   b) For each word encountered going forward:
      ▪ Add an edge to the node consisting of the previous node with the first word removed and the next word appended.
   c) At the end of the text, add an edge to the node consisting of the previous node with the first word removed and the end symbol appended.
2. The edge weights can now be computed by taking the number of edges \((u, v)\) divided by the total number of edges from state \(u\). This last step is strictly not necessary—it will save memory but complicate text generation.

To generate text from a Markov chain, simply do a random walk in the graph from the start state until an end symbol is encountered and record the words passed.

A generalization of the context-free grammar strategy would be to use the same MCMC technique except for word classes and punctuation. This would lead to a dynamically constructed grammar characterizing the structure of the input texts but not the content, but I did not actually implement this.

\(^6\) [http://ucrel.lancs.ac.uk/bncfreq/flists.html](http://ucrel.lancs.ac.uk/bncfreq/flists.html)
5.4.3 Persons

Persons are common in commercial databases, which are often used to track people and their activities. Typical related properties of a person include their age, gender, first name and last name.

The US Census Bureau has published lists of common male first names, female first names and last names, all with cumulative relative frequency in percent with 3 decimal places\(^7\). The data is from their 1990 Census.

I wrote a plug-in that loads these lists and selects random names following the distribution in them. To select such a name:

1. Pick a number \(x\) randomly between 0 and the highest cumulative relative frequency in the file (the cumulative relative frequency of the last name).
2. Perform a binary search for the first name with cumulative relative frequency greater than \(x\). Call the set of words with the same cumulative relative frequency \(W\). This is necessary due to the limited precision of the cumulative relative frequency in the files.
3. Pick a name \(w\) uniformly from \(W\).

5.4.4 Companies

A company is another common entity in commercial databases. They appear as customers (or accounts in sales lingo), as potential customers (leads), competitors, suppliers, etc.

Common properties of companies include name, phone number, fax number, internet domain, contact e-mail address, stock ticker symbol, number of employees, revenue, profits, billing address, shipping address, etc. Some of these are usually vaguely related, such as name, internet domain and stock ticker symbol.

I picked company names either as capitalized strings of 1-3 words (common) or as 3-4 letter acronyms (less common). Then I produced their internet domains by simply concatenating the words of the name and making it lower case and their stock ticker symbols by taking a few letters from the beginning of each word and cutting it down to 4 characters.

5.4.5 US Addresses

US Addresses consist of number, street, city, state and ZIP code. I found a list of ZIP codes with states and cities\(^8\). Then I made up numbers and street names using random words from a dictionary combined with any of the suffixes "Blvd", "Rd", "St", "Ave", "Ln" or "Dr".

5.4.6 Seed Data

Some data is pre-determined and not subject to the randomness and anonymity requirements of other data. Examples include a calendar table of dates with pre-computed weeks, fiscal quarters and other units of time used for bucketing, a table with currency exchange rates by date, and other data that is publically available.

Such data can be directly included in the generated data and does not need special treatment.

5.5 Statistical Analysis of Customer Data

One interesting exercise I did was to try to analyze data from a production database to compute the statistical properties of the foreign keys. A search for ways to fit an unknown parametric distribution to data turned up information on, among other things, the generalized lambda distribution and smooth bootstrap methods.

\(^{7}\) http://www.census.gov/genealogy/names/names_files.html

\(^{8}\) http://www.populardata.com/
I chose to implement a scheme using smooth bootstrapping\(^9\), though possibly not in a statistically rigorous way. To analyze a relationship I compute the cardinality of the relation for each key \(k\) and call it \(c_k\).

To sample this distribution, I first pick a random key \(k'\). Then I let \(s\) be a smoothing term sampled from \(N(0, |\sqrt{c_k'} + 0.5|)\). Then the sample cardinality is \(c_k' + s\).

\(^9\) http://en.wikipedia.org/wiki/Bootstrapping_(statistics)
6 Experiments

This chapter describes the experiments I performed using Le Generator. The main experiment was generating data to fit a subset of the SalesForce.com data model. Additional experiments were performed to evaluate performance and scalability as well as experiments to evaluate the MCMC natural language generator and the smooth bootstrap cardinality simulator.

6.1 Generating SalesForce.com Data

As an example of a real world, commercial data model I picked a subset of the full SalesForce.com data model\(^\text{10}\), version 12. This choice was supported by the fact that I had access to this data from several SalesForce.com accounts and that one of my objectives was to replace this data for testing purposes.

The full schema consists of over 200 tables, but my subset consists of merely 21, with an average of 29 columns each. Most tables have 10-20 columns, but some have more, see Table 6-1. A simplified overview of the relations in this schema is given in Figure 6-1.

<table>
<thead>
<tr>
<th>Table</th>
<th>Number of Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lead</td>
<td>185</td>
</tr>
<tr>
<td>Opportunity</td>
<td>64</td>
</tr>
<tr>
<td>Contact</td>
<td>56</td>
</tr>
<tr>
<td>User</td>
<td>51</td>
</tr>
<tr>
<td>Account</td>
<td>36</td>
</tr>
<tr>
<td>Campaign</td>
<td>34</td>
</tr>
<tr>
<td>OpportunityStage</td>
<td>18</td>
</tr>
<tr>
<td>OpportunityLineItem</td>
<td>16</td>
</tr>
<tr>
<td>RecordType</td>
<td>15</td>
</tr>
<tr>
<td>OpportunityCompetitor</td>
<td>15</td>
</tr>
<tr>
<td>PricebookEntry</td>
<td>14</td>
</tr>
<tr>
<td>UserRole</td>
<td>13</td>
</tr>
<tr>
<td>Product2</td>
<td>13</td>
</tr>
<tr>
<td>Partner</td>
<td>13</td>
</tr>
<tr>
<td>CampaignMember</td>
<td>13</td>
</tr>
<tr>
<td>Pricebook2</td>
<td>11</td>
</tr>
<tr>
<td>OpportunityContactRole</td>
<td>11</td>
</tr>
<tr>
<td>LeadStatus</td>
<td>10</td>
</tr>
<tr>
<td>DatedConversionRate</td>
<td>10</td>
</tr>
<tr>
<td>CurrencyType</td>
<td>9</td>
</tr>
<tr>
<td>CurrentTimestamp</td>
<td>1</td>
</tr>
</tbody>
</table>

*Table 6-1: The number of columns per table in the schema.*

\(^{10}\) http://www.salesforce.com/us/developer/docs/api/index.htm
The purpose of this experiment was to try the framework in an actual testing situation and to evaluate its ease of use, the fitness of the data and its limitations.

I used the scaffolding feature of the framework; see 5.2.2 Second Implementation, to generate empty templates. I proceeded incrementally by first setting the number of records to 100 for each table and generating data with as many primitive types specified as possible.

Then I tried to load the data into a LucidDB warehouse and failed. I specified more dependencies and corrected errors until I could load the data into LucidEra and perform a full ETL with the current version.

I also performed an informal user study where I gave the full framework to an experienced software engineer, motivated by his own need for data, and tried to teach him how to use it.

---

Figure 6-1: A simplified overview of some of the test schema.\[11\]

---

\[11\] Picture downloaded from:
http://www.salesforce.com/us/developer/docs/api/Content/images/Sforce_major_objects.gif
6.2 Performance and Scalability
I also ran a few tests to evaluate the performance of the framework. High performance was not an objective, but knowing how it performs and how well it scales could help answer questions regarding its fitness for particular tasks.

To evaluate performance I took a subset of the SalesForce.com schema described above and fixed the number of records in all the tables except the opportunities table. Then I measured the time taken to generate data. This allowed me to evaluate the scalability of the tool with respect to a single table. It does not take into account the scalability with respect to several tables. Although interesting, designing an experiment to measure this with adequate coverage of all types of relationships was outside the scope of this project.

6.3 Natural Language Generation
I wanted my natural language generator to be able to generate distinct and readable phrases that contained no sensitive company information. I evaluated the MCMC sentence generator by feeding it account descriptions from a production system and generating new phrases. I tried my MCMC implementation of the first, second and third order, and evaluated the results with respect to grammatical correctness and similarity to the original descriptions.

The training set consisted of 10766 descriptions. Each description contained on average 34 words and 220 characters, making the average word length 6 characters.

6.4 Fitting Cardinalities to Production Data
This feature was only briefly tested, by feeding the Account and User tables into the system, analyzing the Account.CreatedById – User.Id foreign key relation, and then generating cardinalities by sampling the smoothed bootstrap distribution. The test data set contained 300 users (184 participating in this relation for a participation of 61%) and 56496 accounts (with the CreatedById always set to a user-id for a NULL-fraction of 0). The output contained 184 users, to match the input and the number of accounts per user was recorded.
7 Results & Limitations

This chapter describes my findings from the experiments described in the previous chapter. I start off with a qualitative analysis of the usability and expressiveness of the framework. Then I get to quantitative performance and scalability analysis. Finally I present some results of the natural language generation experiments.

7.1 Ease of Use

The intended process for using the framework was to first run it in scaffold mode, to get template python modules for each table in the schema. Then open the python templates one by one and add additional metadata and dependency information as required, by selecting and combining the existing generator plug-ins, and using the built-in objects as described in chapter 5. If any types or distributions not already available as generator plug-ins were required, they could be written during this phase.

The subject of my user study, an experienced software engineer, was able to produce quality test data in less than four hours. From a usability standpoint, several issues became clear during this process.

The most obvious one was that the interface was not discoverable. The only way to learn was to look at examples or read documentation or look at the actual generator source code. There was no menu to pick alternatives from—you had to know which ones were available and what they did.

The second problem was that of learning Python—even though it is an easy language to learn, especially the subset needed for data templates, there are still some idioms that could trip up programmers used to other languages.

The third problem was that of name clashes. I had opted to import all names into the data template scope, in order to keep the data definitions as close to regular English as possible. When database field names clashed with generator names, unexpected results became very likely. They could be worked around but there was no automatic detection of this condition and the framework itself gave no clues to the user on how to solve it.

A fourth problem was schema changes. The scaffold feature could only produce empty templates. If the schema changed, there was no way to get the changes into the data templates except manual typing.

The data template for the account template is shown as an example below:

```python
table.number_of_records = 10000
c = Company()
shipping = Address()
billing = choice(shipping, Address())
Id = sfdcId("001")
Name = c.name
Type = choice('Customer', 'Competitor', 'Partner')
Industry = choice('Finance', 'Software', 'Manufacturing')
AnnualRevenue = c.revenue
NumberOfEmployees = c.employees

# Must be created by a user
CreatedById = FK('User', 1)

# Half of the users update accounts
LastModifiedById = FK('User', participation=.5)

# A lot more accounts created lately...
CreatedDate = subtractDays(today(), int(Zipfian(xrange(200))))
LastModifiedDate = mindate(CreatedDate.addDays(N(15, 100)), today())
```
SystemModstamp = mindate(LastModifiedDate.addDays(choice(0, U(0, daysbetween(today(), LastModifiedDate))), today()))

# Want this to be a tree, but cannot express it!
ParentId = None

# Use properties of compound types declared earlier
BillingStreet = billing.number() + " " + billing.street()
BillingCity = billing.city()
BillingState = billing.state()
BillingPostalCode = billing.zip()
BillingCountry = billing.country()
ShippingStreet = shipping.number() + " " + shipping.street()
ShippingCity = shipping.city()
ShippingState = shipping.state()
ShippingPostalCode = shipping.zip()
ShippingCountry = shipping.country()
Phone = c.phone
Fax = c.fax
Website = c.website
TickerSymbol = c.tickersymbol

# Generate some natural language for the description
Description = MCMCPhrase.from_file("generators/accountdescr.mcmcp").sample()

# Owned by any user
OwnerId = FK('User', 1)

# Half of the RecordTypes have ~N(100,50) accounts each
RecordTypeId = FK('RecordType', .5, N(100,50))

7.2 Limitations

Even though I had sacrificed usability in the name of versatility, there were still a few things that simply could not be modeled in an elegant way.

Some complex inter-table relationships, for instance when “File-system-objects” can only have child objects if their “Type” column is set to “Folder”. This is a foreign-key relationship that depends on data in the primary record and cannot be modeled using “Le Generator”. Another example is dependencies involving three or more tables, like a user either creating both accounts and opportunities or none of them.

A frequently occurring pattern in the SalesForce.com data model is the hierarchy: a key relation where the foreign and primary tables are the same. An example is the sales people hierarchy. The user table contains all the people, and each person except the top executive (typically the VP of Sales) has a manager. This type of relationship forms a tree structure with the top executive as root.

Common properties of hierarchies are the number of levels (the depth of the tree) and the number of people per manager at each level.

“Le Generator” could not model hierarchies because there was no way to prevent cycles in the graph. Also there was no way to specify the depth of the tree or the distribution of people in the different levels. This limitation had severe impact on the fitness of the data for use in testing, since hierarchies were frequently used as attributes of dimensions. For example the sales data could be sliced by sales person, sales manager, sales director, or area VP in the LucidEra application and this functionality could not be tested using data provided by “Le Generator”.

Another pattern that is hard to model is found for instance in the currency table. A company has exactly one corporate currency. This would normally be expressed as a foreign key relation from the Company table to the Currency table. In the SalesForce.com data model however, this property is expressed by the column “IsCorporate” in the Currency table. This column contains the value “true” for exactly one record—the corporate currency. Modeling this can be done using the built-in Table object in “Le Generator”, but it is relatively difficult and the resulting data template is complicated.
7.3 Performance
I did all measurements on an Apple Macbook with Intel Core Duo 2 GHz processor and 1GB of RAM running Mac OSX 10.4 “Tiger” and Python 2.5 r25. The results in the chart below indicate that time taken to generate data grows linearly with increasing table size.

Figure 7-1 The time taken to generate data for the test schema depending on the number of opportunities specified.

7.4 Natural Language
A few example phrases are displayed below. I removed identifiable names and brand names to preserve the anonymity of the original data source.

These phrases were generated by the first order MCMC (the grammar checking tool in Microsoft Word was not too happy about these):

- Not qualified. said that improve cost too swamped right now, using a copy of Charity Healthcare outsourced mit Email Boxes Spam an install [brand name removed], recognised brands, with Matt's info to: Exchange im Telekommunikationsbereich ([name removed]), im Einsatz sind.
- The [name removed], Rich ooo [name removed], not be coming through four restaurants to provide comprehensive security team.
- They are looking for email security is [brand name removed], 4 Terabyte Kapazität zur GeldKarte.
- 5,000 mailboxes, they may change) Pains: Spam Filter Mail Frontier As I also selective in large provisioning project to get hold of the manufacture, distribution of superiority, has been on them on your product he's happy to get some background information: 1. Oktober 2003 [brand name removed] on the IT Infrastructure Products and our product and unique in 18 emisoras afiliadas, distribuidas em contato com formulario para este año Dez /06 para desenvolverem o [name removed] ([email address removed]), o Sr. Marcos currently looking at outsourcing solution. Apparently, their own port. See Walmart

These phrases were generated by the second order MCMC:
• quoted through [brand name removed] should move quickly and would be trying to hire *and pay* high market-rate salaries. In addition, [brand name removed] is able to tell me who but they would contact us.

• 1,100 mailboxes. Just migrated all SMTP email gateways sends port25 to [brand name removed] bridgehead or to [brand name removed].

• Not a high technology solutions to leverage technology for CGI and then [brand name removed]’s RBL service goes down, and stop them; and to assist local community school district needs. The company’s solutions encompass consumer, enterprise and storage solutions. Founded in 1853, the firm is headquartered in [city removed], [country removed], operates 12 manufacturing plants — four each in [country removed] and abroad, its 700 employees provide IT-services for the quick response. I think that they are an outsourcing firm for information but if things get worse he will become more serious about anything email sec. related

• Oppt to revisit AS project budgeted for over 300,000 domains and a short duration, let me know how they had already won a contract to keep our operations up and were going to have three layers of AV

• 11/07/06 Spoke to Jeremy he said that he would forward the info to track the aspiring singer’s career via a lead from Andy on this account.

As can be seen above, even the second order MCMC preserves much of the original content. The third order MCMC is even worse and produces verbatim copies of the input descriptions.

The time taken to train the MCMC depends on the order. Timings from an Apple Macbook with Intel Core Duo 2 GHz processor and 1GB of RAM running Mac OSX 10.4 “Tiger” and Python 2.5 r25 are in Table 7-1.

<table>
<thead>
<tr>
<th>Order</th>
<th>User Time [seconds]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>4.5</td>
</tr>
<tr>
<td>2nd</td>
<td>6.2</td>
</tr>
<tr>
<td>3rd</td>
<td>7.4</td>
</tr>
</tbody>
</table>

Table 7-1: The time taken to train an MCMC natural language generator using 10766 phrases.

7.5 Statistical Analysis of Production Data

Results of this experiment are in the chart below. A few things to take note of:

• The generated dataset contains one record with cardinality 0. This means that the number of participating records has gone from 184 to 183.

• Although the interval 10-25 looks slightly over-represented in the generated data the number of records with this cardinality is actually lower than in the original data, but less evenly distributed.

• The total number of accounts turned out to be 33793 in the generated data, considerably less than the 56496 that I started out with.
Figure 7-2 Number of Accounts per User in production data compared to synthetic data. The x-axis (number of accounts) scale is logarithmic.
8 Conclusions and Outlook

The results described in the previous chapter are further analyzed in this chapter and there are some lessons to be learned from this project. In this chapter I will also provide some suggestions for improvement and recommendations for future projects.

8.1 The Good…

"Le Generator” is a flexible and extensible framework for data synthesis. The tool is fast enough to generate databases with hundreds of thousands of records well within a day.

The system can automatically decide the order in which to generate tables, even when there are several intra-table dependencies in the schema.

A number of different types of data can be generated and the framework can easily be extended with more. Data dependencies can be captured in many cases.

Certain numbers, like the average days to close an opportunity, can be set in the data generator and verified in the BI application at the other end of the data pipeline.

8.2 …the Bad…

As described above, the framework is plagued by both usability and versatility limitations. To serve as the primary source of test data, the tool needs to be able to cover all cases. “Le Generator” cannot do that without significant improvements and should therefore only be used as a source of supplemental data.

There is always going to be a trade-off between usability and versatility. My recommendation is to make the priorities clear in the requirements phase, instead of making case-by-case decisions during prototyping.

8.3 …the Ugly

I finish this thesis with a couple of recommendations for improvements of this tool and tips for anyone doing something similar in the future.

Using an imperative language such as Python for meta-data specification makes it hard to process that metadata, compared to a simpler and less powerful declarative language. With a Turing complete language, there is only so much analysis you can do without running into undecidable problems.

In addition using the same language everywhere blurs the lines between the metadata templates, the data dependency handling code and the generator plug-ins. For example, a full name could consist of a given name and a family name—the expression to combine these names should reside in a generator plug-in, but the in the current implementation it could just as well be in the data template. A clear separation of responsibilities should be a goal in any well-engineered software system (McConnel, 2004).

An effort should also be made to separate data about primitive types from data about dependencies. These two pieces of metadata are mixed in the current implementation, which leads to metadata that is hard to read, edit and process.

A user interface should be discoverable. There is no way for a novice user to discover the different featured offered by “Le Generator”. A possible improvement would be a graphical
user interface for meta-data, where the user can interactively select data types from a list of available types.

One source of metadata mentioned in the problem definition is names, such as table names in the schema. This source is completely ignored by the current tool. Having the tool learn what common names mean and intelligently fill fields based on previous specifications would give a huge productivity boost for anyone specifying metadata.

Even though natural language could be generated by the MCMC, an even better approach would be to have a dictionary with words and word classes and let the MCMC learn sentence structures from the input data. This could be extended to also allow it to learn new words if it can recognize the sentence structure surrounding them.

### 8.3.1 A Note on Security

All components of the "Le Generator" framework use unrestricted Python code as their language. As a consequence, running untrusted generators or data templates on your system constitutes a significant risk. A production system intended for widespread deployment should use a sandbox mechanism to restrict the permissions of untrusted modules.
9 Bibliography


