Bijoux: Data Generator for Evaluating ETL Process Quality

Emona Nakuçi Vasileios Theodorou Petar Jovanovic Alberto Abelló

Universitat Politécnica de Catalunya, BarcelonaTech
Barcelona, Spain
emona.nakuci@est.fib.upc.edu, {vasileios,petar,aabello}@essi.upc.edu

ABSTRACT

Obtaining the right set of data for evaluating the fulfillment of different quality standards in the extract-transform-load (ETL) process design is rather challenging. First, the real data might be out of reach due to different privacy constraints, while providing a synthetic set of data is known as a labor-intensive task that needs to take various combinations of process parameters into account. Additionally, having a single dataset usually does not represent the evolution of data throughout the complete process lifespan, hence missing the plethora of possible test cases. To facilitate such demanding task, in this paper we propose an automatic data generator (i.e., *Bijoux*). Starting from a given ETL process model, *Bijoux* extracts the semantics of data transformations, analyzes the constraints they imply over data, and automatically generates testing datasets. At the same time, it considers different dataset and transformation characteristics (e.g., size, distribution, selectivity, etc.) in order to cover a variety of test scenarios. We report our experimental findings showing the effectiveness and scalability of our approach.

Keywords

Data generator; ETL; process quality;

1. INTRODUCTION

Data-centric processes constitute a crucial part of complex business processes responsible for delivering data to satisfy information needs of end users. Besides delivering the right information to end users, data-centric processes must also satisfy various quality standards to ensure that the data delivery is done in the most efficient way, whilst the delivered data are of certain quality level. The quality level is usually agreed beforehand in the form of service-level agreements (SLAs) or business-level objects (BLOs).

In order to guarantee the fulfillment of the agreed quality standards (e.g., data quality, performance, reliability, recoverability, etc.; see [3, 23, 26]), an extensive set of experiments over the designed process must be performed to test the behaviour of the process in a plethora of possible execution scenarios. In essence, the properties of input data play a major role in the resulting quality characteristics of the ETL process. Furthermore, to obtain the finest level of granularity of process metrics, quantitative analysis techniques for business processes (e.g., [9]) propose analyzing the quality characteristics at the level of individual activities and resources.

Obtaining data for running such experiments is often difficult. Sometimes, easy access to the real data is hard, either due to data confidentiality or high data transfer costs. However, in most cases this is due to the fact that only a single instance of data is available, which usually does not represent the evolution of data throughout the complete process lifespan, and hence it cannot cover the variety of possible test scenarios. At the same time, providing synthetic sets of data is known as a labor intensive task that needs to take various combinations of process parameters into account.

In the field of software testing, many approaches (e.g., [8]) have tackled the problem of synthetic test data generation. However, the main focus was on testing the correctness of the developed systems, rather than testing different quality characteristics, which are critical when designing data-centric processes. Moreover, since data-centric processes are typically fully automated, ensuring their correct and efficient execution is pivotal.

In the data warehousing (DW) context, an example of a complex, data intensive and often error-prone data-centric process is the extract-transform-load (ETL) process, responsible for periodically populating a data warehouse (DW) from the available data sources. Gartner has reported in [25] that the correct ETL implementation may take up to 80% of the entire DW project. Moreover, the ETL design tools available in the market [19] do not provide any automated support for ensuring the fulfillment of different quality parameters of the process, and still a considerable manual effort is expected from the designer. Thus we identified the real need for facilitating the task of testing and evaluation of ETL processes in a configurable manner.

In this paper, we revisit the problem of synthetic data generation for the context of ETL processes, for evaluating both the correctness and different quality characteristics of the process design. To this end, we propose an automated data generation algorithm for evaluating ETL processes (i.e., *Bijoux*). Rapidly growing amounts of data represent hidden treasury assets of an enterprise. However, due to dynamic business environments, data quickly and unpre-

DOLAP '14, November 7, 2014, Shanghai, China

http://dx.doi.org/10.1145/2666158.2666183.

dictably evolve, possibly making the software that processes them (e.g., ETL) inefficient and obsolete. Therefore, we need to generate a delicately crafted set of data (i.e., *bijoux*) to test different execution scenarios in an ETL process and detect its behavior over a variety of changing parameters.

To this end, we further tackle the problem of formalizing the semantics of ETL operations and classify them based on the part of the input they access when processing data (i.e., *relation*, *dataset*, *tuple*, *schema*, *attribute*, or *attribute value*) in order to assist *Bijoux* when deciding at which level data should be generated.

Our algorithm, *Bijoux*, is useful during the early phases of the ETL design, when the typical time-consuming evaluation tasks are facilitated with automated data generation. Moreover, *Bijoux* can also assist the complete process lifecycle, enabling easier re-evaluation of an ETL process redesigned for new or changed information and quality requirements. Finally, the *Bijoux*'s functionality for automated generation of syntactic data is also important during the ETL process deployment. It provides users with the valuable benchmarking support (i.e., synthetic datasets) when choosing the right execution platform for their processes.

Outline. The rest of the paper is structured as follows. Section 2 formalizes the notation of ETL processes in the context of data generation and presents a general overview of our approach using an example ETL process. Section 3 formally presents *Bijoux*, our algorithm for the automatic data generation for evaluating ETL processes. In Section 4, we introduce the architecture of the prototype system that implements the functionality of the *Bijoux* algorithm and further report our experimental results. Finally, Section 5 discusses the related work, while Section 6 concludes the paper and discusses possible future directions.

2. OVERVIEW OF OUR APPROACH

2.1 Running example

To illustrate the functionality of our data generation framework, we introduce the running toy example (see Figure 1) that shows a simple ETL process, which matches the first and last name of the customers older than 25 and loads the initials assigned with a surrogate key, to the data warehouse. The example includes several ETL operations. After extracting data from two sources (I_1 and I_2), data are matched with an equi-join (PKey == FKey). Furthermore, the input set is filtered to keep only the persons older than 25 years (Age>25). The first and the last name of each person are then abbreviated to their initials and the unnecessary attributes are projected out. Lastly, data are loaded to the target data store.

The *Bijoux* algorithm thus follows the topological order of the process DAG nodes, (i.e., `I1`, `I2`, `Join`, `Filter`, `Project`, `Attribute Alteration`, and `Load`) and extracts the found flow constraints (e.g., Age>25 or PKey == FKey). Finally, *Bijoux* generates the data that satisfy the given constraints and can be used to simulate the execution of the process.

2.2 Formalizing ETL processes

The modeling and design of ETL processes is a thoroughly studied area, both in the academia [28, 18, 1, 30] and industry, where many tools available in the market [19] often provide overlapping functionalities for the design and execution of ETL processes. Still, however, no particular standard for the modeling and design of ETL processes has been defined, while ETL tools usually use proprietary (platform-specific) languages to represent an ETL process model. To overcome such heterogeneity, *Bijoux* uses the logical (platform-independent) representation of an ETL process, which in the literature (e.g., [30, 14]) is usually represented as a directed acyclic graph (DAG). We thus formalize an ETL process as a DAG consisting of a set of nodes ($\mathbf{V}$), which are either data stores ($\mathbf{DS}$) or operations ($\mathbf{O}$), while the graph edges ($\mathbf{E}$) represent the directed data flow among the nodes of the graph ($v_1 \prec v_2$). Formally:

$ETL = (\mathbf{V}, \mathbf{E})$, such that:
$\mathbf{V} = \mathbf{DS} \cup \mathbf{O}$ and $\forall e \in \mathbf{E} : \exists (v_1, v_2), v_1 \in \mathbf{V} \wedge v_2 \in \mathbf{V} \wedge v_1 \prec v_2$

Data store nodes ($\mathbf{DS}$) in an ETL flow are defined by a schema (i.e., finite list of attributes) and a connection to a source or a target storage for respectively extracting or loading the data processed by the flow.

On the other side, we assume an ETL *operation* to be an atomic processing unit responsible for a single transformation over the input data.

We formally define an ETL flow *operation* as a quintuple:

$o = (\mathbb{I}, \mathbb{O}, \mathbf{X}, \mathbf{P}, \mathbf{A})$, where:

- $\mathbb{I} = \{I_1, \dots I_n\}$ is a finite set of (input) relations.
- $\mathbb{O} = \{O_1, \dots O_m\}$ is a finite set of (output) relations.
- $\mathbf{X}$ ($\mathbf{X} \subseteq \{I_1 \cup I_2 \cup \dots \cup I_n\}$) is a subset of the union of attributes of the schemata $\mathbb{I}$ required by the operation (i.e., *functionality schema*, [27]).
- $\mathbf{P}(\mathbf{X})$ is a predicate over the subset $\mathbf{X}$ of attributes from the input schemata.
- $\mathbf{A}$ is a vector of attributes from the output relation that were added or altered during the operation.

This notation defines the transformations of the input schemata ($\mathbb{I}$) into the result schema ($\mathbb{O}$) by applying predicate $\mathbf{P}$ over input attributes $\mathbf{X}$ and potentially generates or alters attributes in $\mathbf{A}$.

2.3 ETL operation classification

Furthermore, to ensure applicability of our approach to ETL processes coming from major ETL design tools and their typical operations, we performed a comparative study[1] of these tools with the goal of producing a common subset of supported ETL operations. To this end, we considered and analyzed four major ETL tools in the market; two commercial, i.e., Microsoft SQL Server Integration Services (SSIS) and Oracle Warehouse Builder (OWB); and two open source tools, i.e., Pentaho Data Integration (PDI) and Talend Open Studio for Data Integration.

We have noticed that some of these tools (e.g., Pentaho Data Integration) have a very broad palette of specific operations (e.g., PDI has a support for invoking external web services for performing the computations specified by these services). Moreover, some operations can be parametrized to perform different kinds of transformation (e.g., tMap in

[1]More details at: `http://www.essi.upc.edu/~petar/etl-taxonomy.html`

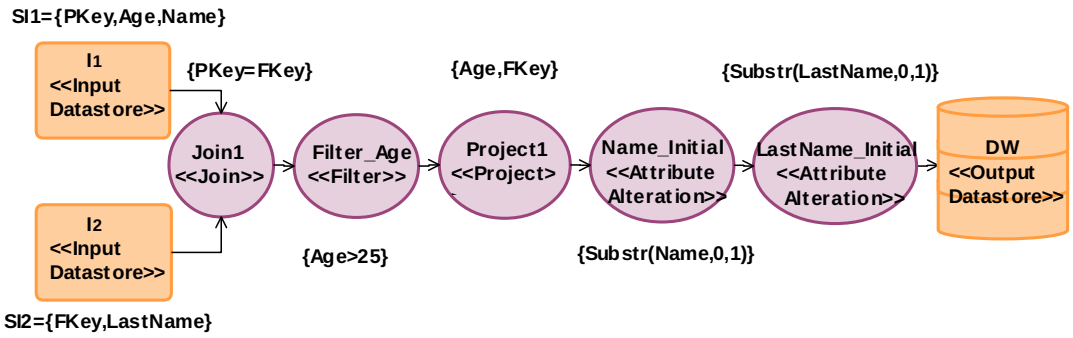

Figure 1: Simple ETL flow example

Table 1: List of operations considered in the framework

Considered ETL Operations	
Aggregation	Join
Attribute Addition	Left Outer Join
Attribute Alteration	Pivoting
Attribute Renaming	Projection
Cross Join	Right Outer Join
Dataset Copy	Router
Datatype Conversion	Sampling
Difference	Sort
Duplicate Removal	Union
Duplicate Row	Union All
Filter	Unpivoting
Intersect	Single Value Alteration

Talend), while others can have overlapping functionalities, or different implementations for the same functionality (e.g., *FilterRows* and *JavaFilter* in PDI). To generalize such a heterogeneous set of ETL operations from different ETL tools, we considered the common functionalities that are supported by all the analyzed tools. As a result, we produced an extensible list of ETL operations considered by our approach (see Table 1).

A similar study of typical ETL operations inside several ETL tools has been performed before in [27]. However, this study classifies ETL operations based on the relationship of their input and output (e.g., *unary*, *n-ary* operations). Such operation classification is useful for processing ETL operations (e.g., in the context of ETL process optimization). In this paper, we further complement such taxonomy for the data generation context. Therefore, as the result, we classify ETL operations based on the part of the relation they access when processing the input data (i.e., *relation*, *dataset*, *tuple*, *schema*, *attribute*, or *attribute value*) in order to assist *Bijoux* when deciding at which level data should be generated. In Figure 2, we show how different parts of a relation are classified, which forms the basis for our ETL operation classification.

We further define the semantics of ETL operations using the notation introduced above, i.e., $o = (\mathbb{I}, \mathbb{O}, \mathbf{X}, \mathbf{P}, \mathbf{A})$.

An ETL operation processes input relations $\mathbb{I}$, hence based on the classification in Figure 2, the semantics of an ETL operation should express transformations at (1) the *schema* (i.e., generated/projected-out schema), (2) the *tuple* (i.e., passed/modified/generated/removed tuples), and (3) the *dataset* level (i.e., output cardinality).

We further give an example of formalizing the semantics

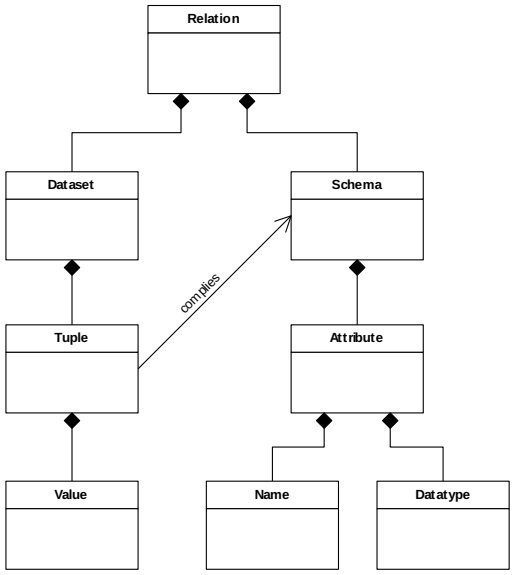

Figure 2: Relation-access based classification

of an ETL operation (*SingleValueAlteration*[2]) that follows previously described notation.

$\forall$(I,O,X,$\mathbb{S}$,A) (F(I,O,X,$\mathbb{S}$,A) $\rightarrow$ (SO=SI $\wedge$ |O|=|I|))
$\forall t_i \in I$ ($S_1(t_i[X])$ $\rightarrow$
$\exists t_o \in O$ (t_o[SO $\setminus$ A]=t_i[SI $\setminus$ A] $\wedge$ t_o(A)=$S_2(t_i[X])$))

2.4 Bijoux overview

Intuitively, starting from a logical model of an ETL process and the semantics of ETL operations, *Bijoux* analyzes how the attributes of input data stores are restricted by the semantics of the ETL process operations (e.g., *filter* or *join* predicates) in order to generate the data that satisfy these restrictions. To this end, *Bijoux* moves iteratively through the topological order of the nodes inside the DAG of an ETL process and extracts the semantics of each ETL operation to analyze the constraints that the operations imply over the input attributes. At the same time, *Bijoux* also follows the constraints' dependencies among the operations to simultaneously collect the necessary parameters for generating data for the correlated attributes (i.e., *value ranges*, *datatypes*, and *the sizes of generated data*). Using the collected parameters, *Bijoux* then generates sufficient sets of

[2] *SingleValueAlteration* operation defines the functionality of modifying the attribute from the input schemata of an operation if a given condition is satisfied.

input data to satisfy all found constrains, i.e., to simulate the execution of the complete data flow. The algorithm can be additionally parametrized to support data generation for different execution scenarios.

Typically, an ETL process should be tested for different sizes of input datasets (i.e., different *scale factors*) to examine its scalability in terms of growing data. Importantly, *Bijoux* is extensible to support data generation for different characteristics of input datasets (e.g., *size*), attributes (e.g., *value distribution*) or ETL operations (e.g., *operation selectivity*). We present in more detail the functionality of our data generation algorithm in the following section.

3. DATA GENERATION ALGORITHM

The *Bijoux* algorithm (see Algorithm 1) explores the input logical model of an ETL process (ETL), extracts the flow constraints, as well as other generation parameters at the level of attributes and ETL operations and generates the data to correspond to these parameters.

In particular, the algorithm includes three main stages (i.e., 1 - *extraction*, 2 - *analysis*, and 3 - *data generation*).

Example. For illustrating the functionality of our algorithm, we will use the running example introduced in Section 2.1. Input datastores of the example ETL flow are: $\mathbb{I}$ = {I_1, I_2}, with schemata SI_1 = {PKey, Age, Name} and SI_2 = {FKey, LastName}; whilst the topological order of their nodes is: {Join, Filter, Project, Attribute Alteration1, Attribute Alteration2}. □

Before going into detail about the three steps of *Bijoux*, we present the main structures maintained by the algorithm. While analyzing the given ETL process model, *Bijoux* keeps three structures for recording different parameters that potentially affect the *data generation* process:

- *Constraints Matrix* (TC). Two-dimensional array that for each attribute (rows) of the input data stores, and each operation (columns) of the input ETL process, contains a set of constraints that the given ETL operation implies over the given input attribute (e.g., *constraint(Age,Filter)* = $Age > 25$; see Figure 3).

- *Attribute parameters* (AP). An array of the data generation parameters at the level of individual attributes of input data stores of the ETL process. An element of this array contains information about the considered attribute (i.e., attribute name, attribute datatype, attribute property list). Attribute property list further contains an extensible list of attribute properties that should be considered during data generation (e.g., *distribution(PKey)* = *uniform*; see Figure4:left).

- *Operation parameters* (OP). An array of the data generation parameters at the level of ETL operations, extracted from an input ETL flow or defined by the user. An element of this array contains information about the considered ETL operation (i.e., operation name, operation property list). Operation property list further contains an extensible list of operation or quality properties that should be considered during data generation (e.g., *operation_selectivity(Filter)* = *0.7*; see Figure 4:right).

In what follows, we discuss the three main stages of our data generation algorithm. Notice that the first stage (*extraction*) processes the complete ETL process to extract necessary generation parameters and fill the above mentioned structures (i.e., AP, OP, and TC). The *analysis* and *data generation* stages further use these structures to generate data for each attribute of the input data stores.

1. *Extraction* stage (Steps 1 - 12) starts from the logical model of an ETL process (ETL) and first obtains the source data stores from the process DAG (Step 2). The algorithm then for each attribute of the source data stores (i.e., $a[i]$; Step 5) and each ETL operation in a topological order of its node in a DAG (i.e., $o[j]$; Step 7) extracts the data generation parameters, i.e., Steps 6 and 8, respectively. At the same time, this stage extracts the semantics of each operation $o[j]$ and searches for the constraints that the operation implies over the given attribute $a[i]$ (i.e., $c[i, j]$; Step 9). As a result, *extraction* stage generates the above mentioned structures (i.e., AP, OP, and TC) used throughout the rest of the algorithm.
   Example. *Bijoux* first iterates through the set of 5 attributes of input schemata ($I_1 = \{PKey, Age, Name\}$ and $I_2 = \{FKey, LastName\}$) and extracts parameters at the attribute level (i.e., attribute datatype and value distribution; see the extracted values in Figure 4:left). Moreover, for each ETL operation of the example ETL flow in Figure 1, in the topological order of their nodes (i.e., *Join, Filter, Project, Attribute Alteration1, Attribute Alteration2*), *Bijoux* collects the parameters at the operation level (i.e., operation selectivity; see the extracted values in Figure 4:right). At the same time, for each operation and each attribute affected by that operation, *Bijoux* stores the semantics of that operation in TC (see Figure 3).

2. *Analysis* stage (Steps 13 - 42) is responsible for iterating over each attribute of the generated structures and analyzing how the collected parameters (i.e., AP and OP; Steps 18 and 20) affect our data generation process. For each attribute (i.e., ith row of TC), we store the information used during the data generation stage (e.g., datatype, attribute properties) inside the gP_i structure, as well as the value ranges. In a typical scenario, a single ETL operation may apply constraints over multiple attributes from the input. Thus, the data for these *dependent attributes* (i.e., the attributes included in a single ETL operation constraint; e.g., *Join[PKey = FKey]* in Figure 3) must be simultaneously generated. To this end, after analyzing data generation parameters of a single attribute for a single operation, we must follow the list of all *dependent attributes* from the given operation (Step 21), and ana-

		Flow Operations				
		Join	Filter	Project	Attribute Alteration	Attribute Alteration
Input Schemata	PKey	{PKey=FKey}				
	Age		{Age>25}	{Age,FKey}		
	Name				{Substr(Name,0,1)}	
	FKey	{PKey=FKey}		{Age,FKey}		
	LastName					{Substr(LastName,0,1)}

Figure 3: Data gen. parameters (Constraints Matrix - TC)

	Datatype	Distribution
PKey	Long	Uniform
Age	Integer	Normal(30,5)
Name	String	NA
FKey	Long	Uniform
LastName	String	NA

Attribute Parameters (AP)

	Selectivity
Join	0.6
Filter	0.7
Project	1
Attribute Alteration	1
Attribute Alteration	1

Operation Parameters (OP)

Figure 4: Data gen. parameters (AP and OP)

	Join	Filter	Project	Attribute Alteration	Attribute Alteration
PKey	{PKey=FKey}				
Age		{Age>25}	{Age,FKey}		
Name				{Substr(Name,0,1)}	
FKey	{PKey=FKey}		{Age,FKey}		
LastName					{Substr(LastName,0,1)}

Figure 5: TC during the first iteration of the analysis stage

lyze data generation parameters for these attributes in the same manner (Steps 24, 26, and 27). Similarly, we analyze operation constraint semantics. Based on the operation constraints, we find the range (lower and upper limit) of each attribute value (and dependent ones) and update it accordingly whenever the same attribute is encountered in the following operations (Steps 27, 28 and 36). The idea of ranges has a broad spectrum of applicability, because it can be applied to numerical attributes as well as data and textual ones. Later, these ranges will drive the data generation stage. At the end of this stage, the *genParams* list contains the information for all the dependent attributes, i.e., the attributes for which the data should be simultaneously generated.

Example. In the first iteration of the analysis stage, *Bijoux* analyzes the operation semantics for the first (`PKey`) attribute (i.e., the first row from the *TC* matrix). For the *PKey* attribute *Bijoux* finds the constraint "*PKey* = *FKey*", and by analyzing it, it discovers that it also contains a *dependent attribute FKey*. Therefore, the algorithm then iterates not only over the operations that use *PKey* in their semantic, but also those that use the *dependent attribute FKey* (see Figure 5). *Bijoux* continues further to collect all the constraints of operations that use the next attribute (i.e., `Age`), and finds Filter (*Age* > 25) and Project that use this attribute. *Bijoux* thus updates the range of `Age` (Step 36) and sets the lower limit to 25, as we need to generate values greater than 25. The algorithm continues in the same manner for the rest of the unprocessed attributes. □

3. *Data generation* stage (Steps 44 - 47), finally, uses the generation parameters (*genParams*), resulted from the analysis stage and the ranges information, and generates data to satisfy all the restrictions extracted from the input ETL process (*ETLFlow*), (i.e., Step 45). As discussed before, data generation process can be further parametrized with additional information (e.g., the scale factor of the generated dataset - size).

Example. For the `FKey` and `PKey` attributes, *Bi-*

Algorithm 1 Bijoux Algorithm

Input: ETL, size
Output: genData

```
 1: AP ← ∅; OP ← ∅; TC ← ∅;
 2: 𝔻𝕊 ← SourceNodes(ETL);
 3: for each DS ∈ 𝔻𝕊 do
 4:     SI ← InputSchema(DS);
 5:     for each attribute a[i] ∈ SI do
 6:         AP[i] ← Extract(a[i]);
 7:         for each operation o[j] ∈ TopOrder(ETL) do
 8:             OP[j] ← Extract(o[j]);
 9:             TC[i,j] ← Extract(c[i,j]);
10:         end for
11:     end for
12: end for
13: visited ← Boolean Array[Attributes(TC)] {false};
14: for (i := 1 to Rows(TC)) do
15:     if (!visited[i]) then
16:         visited[i] ← true; genParams ← φ;
17:         SetRange(range_i,DefBounds(datatype_i));
18:         gP_i ← Analyze(AP[i]);
19:         for (j := 1 to Operations(TC)) do
20:             Update(gP_i, Analyze(OP[j]));
21:             for each k ∈ DepAttrsIndexes(TC[i,j]) do
22:                 visited[k] ← true;
23:                 SetRange(range_k,DefBounds(datatype_k));
24:                 gP_k ← Analyze(AP[k]);
25:                 for (l := 1 to Columns(TC)) do
26:                     Update(gP_k, Analyze(OP[l]));
27:                     UpdateRange(range_k,TC[k,l]);
28:                     UpdateRange(range_i,TC[k,l]);
29:                     if (isSelectivityRequired) then
30:                         UpdateRangeInv(range_{k_inv},TC[k,l]);
31:                     end if
32:                     Add(genParams,
33:                         <gP_k,range_k,range_{k_inv}>);
34:                 end for
35:             end for
36:             UpdateRange(range_i,TC[i,j]);
37:             if (isSelectivityRequired) then
38:                 size_1 ← Calculate(OP[j],size);
39:                 size_2 ← CalculateInv(OP[j],size);
40:                 UpdateRangeInv(range_{i_inv},TC[i,j]);
41:             end if
42:         end for
43:         Add(genParams,<gP_i,range_i,range_{i_inv}>);
44:         for each <gP,range,range_inv>∈ genParams do
45:             genData ← GenData(gP,range,size_1) ∪
46:                       GenData(gP,range_inv,size_2);
47:         end for
48:     end if
49: end for
```

joux previously found the constraint that these two attributes need to have equal values to satisfy the join condition (see Figure 5). Therefore, since both attributes are long integers with a uniform distribution (see Figure 4:left) and if the user has specified the size of the input load to be 100, *Bijoux* generates 100 numerical long values equal among them using the uniform distribution generator. Notice that here we do not consider specific ranges of the generated values for `FKey` and `PKey` since only the condition that they are mutually equal is sufficient. On the other side, the generated integer values for the `Age` attribute must satisfy the range defined in the previous stage, i.e., $(25, MAX\_INT)$, and follow the normal value distribution.

The above description has covered the general case of data generation without considering other generation parameters. However, given that our data generator aims at generating data to satisfy other configurable parameters, we illustrate here as an example the adaptability of our algorithm to the problem of generating data to additionally satisfy *operation selectivity*. To this end, the algorithm now also analyzes the parameters previously stored in OP (see Figure 4:right).

From the extracted OP (see Figure 4:right), *Bijoux* finds that the *Filter* operation has a selectivity of *0.7*. While iterating over the TC matrix (see Figure 3), *Bijoux* extracts operation semantics for *Filter* and finds that it uses the attribute *Age* (*Age* > *25*). With the selectivity factor of *0.7* from OP, *Bijoux* infers that out of all incoming tuples for the Filter, 70% should satisfy the constraint that *Age* should be greater than 25, while 30% should not (i.e., 30% of *Age* values should be less or equal to 25). We analyze the selectivity as follows:

- To determine the total number of incoming tuples for *Filter*, we consider preceding operations, which in our case is *Join* with selectivity *0.6*. Considering the input load size is 100, this means that in total $0.6 * (max(100, 100)) = 60$ tuples pass the join condition.
- From these 60 tuples only 70%, based on the *Filter* selectivity, (i.e., 42 tuples) will successfully pass both the filtering (*Age* > *25*) and the join condition (*PKey* = *FKey*).
- The remaining of 18 tuples should fail (i.e., *Age* ≤ *25*). In order to generate the data that do not pass this operation of the flow, we rely on the inverse constraints that we parse from the algorithm (Steps 30 and 40).

Finally, after *Bijoux* collected and analyzed information from TC (*Age* > *25*; see Figure 3), AP (long value normally distributed with mean 30 and standard deviation 5; see Figure 4:left) and OP (selectivity *0.7*; see Figure 4:right), it proceeds to the data generation stage. Similarly, since the *Join* operation precedes *Filter*, *Bijoux* must also consider its semantics (i.e., *PKey* = *FKey*). Its respective parameters from AP suggest long numerical values having a uniform distribution.

As a result of the above analysis, we need to generate a dataset (I_1 and I_2) such that the output of the *Join* operation is 60 tuples that satisfy join condition, out of which, 42 have *Age* greater than 25, while the rest have *Age* smaller or equal to 25.

3.1 Enhanced ETL flow example

The running example of the ETL flow that we have used so far is expressive enough to illustrate the functionality of our framework, but it appears too simple to showcase the benefits of our approach. In this respect, we introduce here a more complex ETL flow (Fig. 7), which performs the same task as the simple example, but ensures greater data quality of the output data.

To this end, this ETL flow includes an additional filter operation that checks for null values of a specific attribute (i.e., Age) and filters out corresponding tuples, thus improving data completeness. Furthermore, it includes one additional input data store (i.e., Alt_DS) that acts as an alternative data source in order to cross-check data and improve their consistency and completeness, by filling in missing values. Thus, the *Join* of the alternative data store to the flow is followed by a *Value Alteration* operation to perform the required calculations and the added attributes, which are then no longer required for subsequent operations, are added to the list of projected out attributes of the following *Project* operation.

Using our algorithm, as we have defined above, we can generate input data not only to test the ETL process flow, but also to obtain feedback that can guide the process configuration and tuning to satisfy non-functional requirements. This is possible due to the use of configurable set of parameters during the data generation process and the use of adequate measures for the evaluation of quality characteristics during process simulation [26].

In our example, we can configure the *size* of the generated data for the alternative data store, resulting in variable matching of datasets to fill in missing values and thus different levels of data quality. Based on measured data quality and given related business requirements, this can allow for the identification of the right size of alternative data stores, relative to the size of existing data sources, which in real use case scenarios can have important implications on the cost.

Another configurable parameter for data generation is *distribution*. Using different distribution properties in our example, we can adjust the matching between data stores, providing similar analysis capabilities with the previous example. In addition, we can configure the distribution of null values, enabling the assessment of the performance of the *Filter_Null* operation.

When it comes to *operation selectivity*, it can be configured for the *Join2* operation as well as for the *Filter_Null* operation, resulting in generated input data of variable properties that impose differences in the measured performance and data quality of the ETL process.

We have explained above how the parametrization of our input data generation enables the testing and tuning of an ETL process with respect to data quality and performance. Essentially, the same ETL flow can be simulated using different variations of the data generation properties and the measured quality characteristics will indicate the best models. Similarly, other quality characteristics can be considered, for example reliability and recoverability, by adjusting the distribution of input data that result to exceptions and the selectivity of exception handling operations.

4. EXPERIMENTS

In this section, we report the experimental findings, after

	Generated Data			
PKey	45	125	97	3
Age	53	34	26	44
Name	"Ayline"	"Robert"	"Joseph"	"Trevor"
FKey	45	125	97	3
LastName	"Scott"	"Smith"	"Moore"	"Woody"

Data generated after analyzing the last ETL operation

I1

Generated Data				
PKey	45	125	97	3
Age	53	34	26	44
Name	"Ayline"	"Robert"	"Joseph"	"Trevor"

Result Set

PKey	Age	Name
45	53	"Ayline"
125	34	"Robert"
97	26	"Joseph"
3	44	"Trevor"

I2

FKey	45	125	97	3
LastName	"Scott"	"Smith"	"Moore"	"Woody"

FKey	LastName
45	"Scott"
125	"Smith"
97	"Moore"
3	"Woody"

Resulting generated datasets

Figure 6: Generated datasets

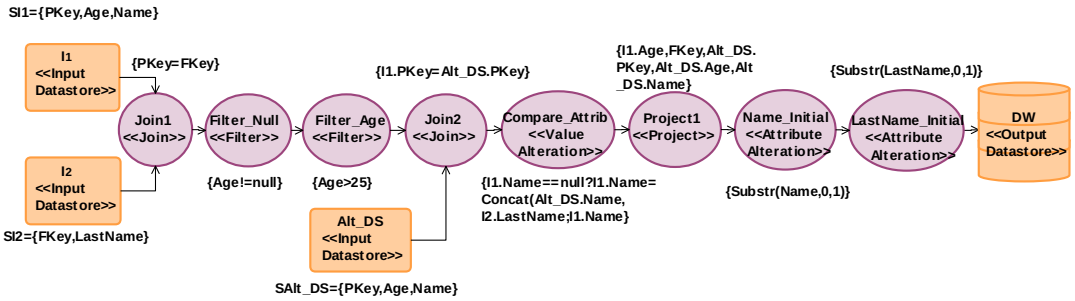

Figure 7: Enhanced ETL flow example

scrutinizing different performance parameters of *Bijoux*, by using the prototype that implements its functionalities.

We first introduce the architecture of a prototype system that implements the functionality of the *Bijoux* algorithm.

Input. The main input of the *Bijoux* framework is an ETL process. As we previously discussed, we consider that ETL processes are provided in the logical (platform-independent) form, following previously defined formalization, and carrying different parameters that can lead the process of data generation (e.g., *attribute distribution*, *operation selectivity*). Beside, data generation parameters can be also defined separately (e.g., *scale factors*).

***Bijoux*'s architecture.** The *Bijoux*'s prototype is modular and based on the layered architecture. The three main layers implement the core functionality of the *Bijoux* algorithm (i.e., *extraction*, *analysis*, and *data generation*), while the additional two layers are responsible for communicating with the outside world, for importing ETL flows and exporting obtained results back to the user. Notice that while the former three layers are integral part of our tool, the latter two can be externally provided and plugged to our framework (e.g., flow import plugin [14]).

Output. The output of our framework is the collection of datasets generated for each input data store of the ETL process. These datasets are generated to satisfy the constraints extracted from the flow as well as the parameters gathered from the process description (i.e., distribution, operation selectivity, load size).

4.1 Experimental setup

Here, we focused on testing both the functionality and correctness of the *Bijoux* algorithm discussed in Section 3, and different quality aspects, i.e., data generation overhead (*performance*) and *scalability* wrt. the growing complexity of both the ETL design and input load sizes. We performed the performance testing considering the several ETL test cases, which we describe in what follows.

Our experiments were carried under a Windows 32-bit machine, Processor Core 2 Duo, 2.1 GHz and 4GB of RAM. The test cases consider a subset of ETL operations, i.e., *Join, Filter, Attribute Addition* and *Project*. Starting from this basic scenario, we use the implemented flow generation scripts to automatically create other, more complex, synthetic ETL flows. To this end, we incrementally replicate the existing ETL operations and add them to the given ETL flow. The motivation for building these flow generation scripts comes from the fact that obtaining the real world set of ETL flows covering different scenarios with different complexity and load sizes is hard and often impossible.

Basic scenario. The basic scenario contains two input datastores I_1, I_2, and the considered operations are *Join, Filter, Project, Attribute Addition*. So in total, we have four operations present in the flow.

Scenarios creation. Starting from this basic scenario, we create more complex ETL flows by adding additional operations, i.e., *Join, Filter* in various positions of the original flow.

We iteratively create 6 cases of different ETL flow complexities and observe the *Bijoux*'s execution times for these cases, starting from the basic ETL flow:

- *Case 1.* Basic ETL scenario, consisting of four operations, i.e., *Join, Filter, Project, Attribute Addition*, as described above.
- *Case 2.* ETL scenario consisting of 5 operations, starting from the basic one and adding an additional *Filter*

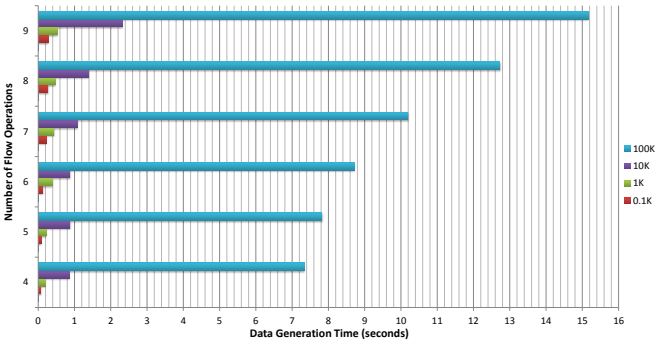

Figure 8: Generation time wrt. the flow complexity

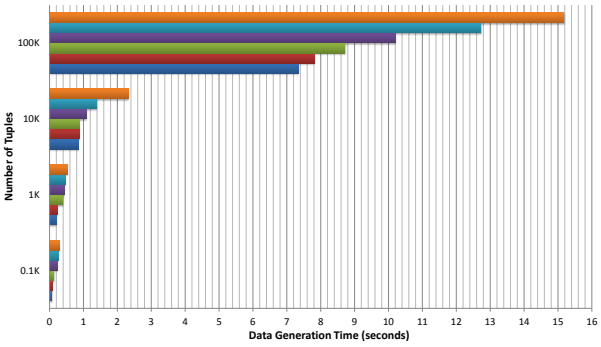

Figure 9: Generation time wrt. the load size

operation to the flow.

- *Case 3.* ETL scenario consisting of 6 operations, starting from the basic one and adding an additional *Join* operation to the flow. When adding a *Join* operation we also add a *Project* and an *Input datastore* (replicated from the existing one), in order to guarantee matching schemata.
- *Case 4.* ETL scenario consisting of 7 operations. Additional *Join* and *Filter* operations are added to the basic flow choosing the random but correct position on the fly.
- *Case 5.* ETL scenario consisting of 8 operations, starting from the basic scenario and adding *Join* and two *Filter* operations.
- *Case 6.* ETL scenario consisting of 9 operations. Two additional *Join* operations along with a *Filter* operations are added to the basic flow.

4.2 Experimental results

We measure the execution time of the data generation process for the above 6 scenarios covering different ETL flow complexities. For each scenario we run *Bijoux* and generate 4 different datasets (i.e., with different load size). The load size is represented with the number of generated tuples per each input datastore of the flow.

- 100 (0.1K) generated tuples
- 1,000 (1K) generated tuples
- 10,000 (10K) generated tuples
- 100,000 (100K) generated tuples

Figure 8 illustrates the increase of data generation time when moving from the simplest ETL scenario to a more complex one, while keeping the load size constant (i.e., the bars of the same color represent generation time for the same load size with different ETL flow complexities). In addition, it also shows the increasing generation time for the load sizes from 100 to 100,000 tuples for a single ETL flow complexity. Importantly, notice that while the increase of the time wrt. the growing ETL flow complexity shows to be tractable, the margin of increasing time wrt. the size of generated data is much higher, indicating the exponential complexity of the data generation problem in terms of the size of generated

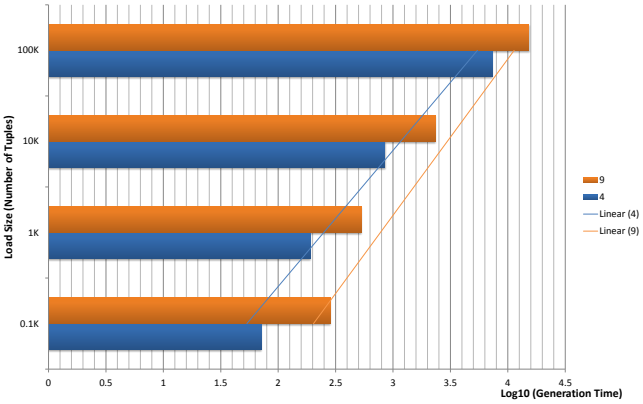

Figure 10: Linear trend of the data generation time for the extreme ETL complexities (cases 1 and 6) wrt. the increasing load size

data. The similar exponential trend is also observed in Figure 9, where we analyze the execution time for different load sizes. The growing trend of the execution time wrt. to the ETL flow complexity is justified by the fact that the semantics of the flow are increasing in number of operation and complexity, imposing more rules and constraints over the generated data.

Regarding the exponential correlation of the execution time to the growing load size, we perform the additional analysis where we proportionally scaled the differences in the load size and the corresponding execution time by using the logarithmic scale (see Figure 10). Interestingly, we have observed the linear trend of the generation time wrt. the load size in this case, for the two extreme scenarios under the study: *case 1.* for the ETL flow with 4 operations, and *case 6.* for the ETL flow with 9 operations. On the additional note, the performance shows a linear trend, which indicates further scaling up opportunities by means of parallelizing data generation tasks for independent datasets.

5. RELATED WORK

The problem of constraint-guided generation of synthetic data has been previously studied in the field of software testing [8]. The context of this work is the mutation analysis of software programs, where for a program, there are several "mutants" (i.e., program instances created with small, incorrect modifications from the initial system). The approach

analyzes the constraints that "mutants" impose to the program execution and generates data to ensure the incorrectness of modified programs (i.e., "to kill the mutants"). This problem resembles our work in a way that it analyzes both the constraints when the program executes and when it fails to generate data to cover both scenarios. However, this work mostly considered generating data to test the correctness of the program executions and not its quality criteria (e.g., performance, recoverability, reliability, etc.).

Moving toward the database world, [31] presents a fault-based approach to the generation of database instances for application programs, specifically aiming to the data generation problem in support of white-box testing of embedded SQL programs. Given an SQL statement, the database schema definition and tester requirements, they generate a set of constraints which can be given to existing constraints solvers. If they are satisfiable, a desired database instances are obtained. Similarly, for testing the correctness of relational DB systems, a study in [7] proposes a semi-automatic approach for populating the database with meaningful data that satisfy database constraints. Work in [2] focuses on a specific set of constraints (i.e., cardinality constraints) and introduces efficient algorithms for generating synthetic databases that satisfy them. Unlike the previous attempts, in [2], they generate synthetic database instance from scratch, rather then by modifying the existing one. Furthermore, in [5], they propose a query-aware test database generator called QAGen. The generated database satisfies not only constraints of database schemata, table semantics, but also the query along with the set of user-defined constraints on each query operator. Other work [12] presents a generic graph-based data generation approach, arguing that the graph representation supports the customizable data generation for databases with more complex attribute dependencies. The approach most similar to ours [15] proposes a multi-objective test set creation. They tackle the problem of generating branch-adequate test sets, which aims at creating test sets to guarantee the execution of each of the reachable branches of the program. Moreover, they model the data generation problem as a multi-objective search problem, focusing not only on covering the branch execution, but also on additional goals the tester might require, e.g., memory consumption criterion. However, the above works focus solely on relational data generation by resolving the constraints of the existing database system. Our approach follows this line but in a broader way, given that *Bijoux* is not restricted to relational schema and is able to tackle more complex constraint types, not supported by the SQL semantics. In addition, we do not generate a single database instance, but rather the heterogeneous datasets based on different information (e.g., input schema, data types, distribution, etc.) extracted from the ETL flow.

Both research and industry are particularly interested in benchmarking ETL and in general integration processes in order to evaluate process designs and compare different integration tools (e.g., [22, 6]). Both works note the lack of a widely accepted standard for evaluating integration processes. The former work focuses on defining a benchmark at the logical level of data integration processes, meanwhile assessing optimization criteria as configuration parameters. Whereas, the later works at the physical level by providing a multi-layered benchmarking platform called DIPBench used for evaluating the performance of data integration systems. These works also note that an important factor in benchmarking data integration systems is defining similar workloads while testing different ETL scenarios to evaluate ETL flows and measure satisfaction of different quality objectives. These approaches do not provide any automatable means for generating benchmark data loads, while their conclusions do motivate our work in this direction.

Some approaches have been working on providing data generators that are able to simulate real-world data sets for the purpose of benchmarking and evaluation. [10] presents one of the first attempts of how to generate synthetic data used as input for workloads when testing the performance of database systems. They mainly focus on the challenges of how to scale up and speed up the data generation process using parallel computer architectures. In [17], the authors present a tool called Big Data Generator Suite (BDGS) for generating Big Data meanwhile preserving the 4V characteristics of Big Data [3]. BDGS is part of the BigDataBench benchmark [29] and it is used to generate textual, graph and table structured datasets. BDGS uses samples of real world data, analyzes and extracts the characteristics of the existing data to generate loads of "self-similar" datasets. In [21], the parallel data generation framework (PDGF) is presented. PDGF generator uses XML configuration files for data description and distribution and generates large-scale data loads. Thus its data generation funcionalities can be used for benchmarking standard DBMSs as well as the large scale platforms (e.g., MapReduce platforms). Other prototypes (e.g., [11]) offer similar data generation functionalities. In general, this prototype allows *inter-rows*, *intra-rows*, and *inter-table* dependencies which are important when generating data for ETL processes as they must ensure the multi-dimensional integrity constraints of the target data stores. The above mentioned data generators provide powerful capabilities to address the issue of generating data for testing and benchmarking purposes for database systems. However, they are not particularly tailored for ETL-like processes, i.e., the data generation is not led by the constraints that the process operation imply over the data.

Lastly, given that the simulation is a technique that imitates the behavior of real-life processes, and hence represents an important means for evaluating processes for different execution scenarios [20], we discuss several works in the field of simulating business processes. Simulation models are usually expected to provide a qualitative and quantitative analysis that are useful during the re-engineering phase and generally for understanding the process behavior and reaction due to changes in the process [16]. [4] further discusses several quality criteria that should be considered for the successful design of business processes (i.e., *correctness*, *relevance*, *economic efficiency*, *clarity*, *comparability*, *systematic design*). However, as shown in [13] most of the business process modeling tools do not provide full support for simulating business process execution and the analysis of the relevant quality objectives. We take the lessons learned from the simulation approaches in the general field of business processes and go a step further focusing our work to data-centric (i.e., ETL) processes and the quality criteria for the design of this kind of processes [24, 26].

[3] *volume*, *variety*, *velocity* and *veracity*

6. CONCLUSIONS AND FUTURE WORK

In this paper, we study the problem of synthetic data generation in the context of multi-objective evaluation of ETL processes. We propose an ETL data generation framework (*Bijoux*), which aims at automating the parametrized data generation for evaluating the correctness of ETL process models, as well as a set of different quality criteria (e.g., reliability, recoverability, freshness, etc.), ensuring both accurate and efficient data delivery. Thus, beside the semantics of ETL operations and the constraints they imply over input data, *Bijoux* takes into account different quality-related parameters, extracted or configured by an end-user, and guarantees that generated datasets fulfill the restrictions implied by these parameters (e.g., operation selectivity).

We have evaluated the feasibility and scalability of our approach by prototyping our data generation algorithm. The first experimental results have shown a linear (but increasing) behavior of *Bijoux*'s overhead, which suggests that the algorithm is potentially scalable to accommodate more intensive tasks. At the same time, we have observed different optimization opportunities to scale up the performance of *Bijoux*, considering both the higher complexity of ETL flows and larger volumes of generated data. We further plan to extend the set of ETL operations supported by *Bijoux*.

As another immediate future step, we consider testing the framework for covering a broader spectrum of configurable parameters, especially focusing on the ones that enable the evaluation of different quality criteria of an ETL process. Moreover, we plan on additionally validating and exploiting the functionality of this approach in the context of quality-driven ETL process design. Finally, following the experimental observations, we plan on adapting *Bijoux* to support scaling up the data generation process, by means of parallelizing independent data generation tasks.

7. ACKNOWLEDGMENTS

This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College" (IT4BI-DC). This work has also been partly supported by the Spanish Ministerio de Ciencia e Innovación under project TIN2011-24747.

8. REFERENCES

[1] Akkaoui, Z.E., Zimányi, E., Mazón, J.N., Trujillo, J.: A bpmn-based design and maintenance framework for etl processes. IJDWM 9(3), 46–72 (2013)

[2] Arasu, A., Kaushik, R., Li, J.: Data generation using declarative constraints. In: SIGMOD Conference. pp. 685–696 (2011)

[3] Barbacci, M., Klein, M.H., Longstaff, T.A., Weinstock, C.B.: Quality attributes. Tech. rep., CMU SEI (1995)

[4] Becker, J., Kugeler, M., Rosemann, M.: Process Management: a guide for the design of business processes: with 83 figures and 34 tables. Springer (2003)

[5] Binnig, C., Kossmann, D., Lo, E., Özsu, M.T.: Qagen: generating query-aware test databases. In: SIGMOD Conference. pp. 341–352 (2007)

[6] Böhm, M., Habich, D., Lehner, W., Wloka, U.: Dipbench toolsuite: A framework for benchmarking integration systems. In: ICDE. pp. 1596–1599 (2008)

[7] Chays, D., Dan, S., Frankl, P.G., Vokolos, F.I., Weber, E.J.: A framework for testing database applications. In: ISSTA. pp. 147–157 (2000)

[8] DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Trans. Software Eng. 17(9), 900–910 (1991)

[9] Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)

[10] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly Generating Billion-Record Synthetic Databases. In: SIGMOD Conference. pp. 243–252 (1994)

[11] Hoag, J.E., Thompson, C.W.: A parallel general-purpose synthetic data generator. SIGMOD Record 36(1), 19–24 (2007)

[12] Houkjær, K., Torp, K., Wind, R.: Simple and realistic data generation. In: VLDB. pp. 1243–1246 (2006)

[13] Jansen-vullers, M.H., Netjes, M.: Business process simulation: a tool survey. In: In Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN (2006)

[14] Jovanovic, P., Simitsis, A., Wilkinson, K.: Engine independence for logical analytic flows. In: ICDE (2014)

[15] Lakhotia, K., Harman, M., McMinn, P.: A multi-objective approach to search-based test data generation. In: GECCO. pp. 1098–1105 (2007)

[16] Law, A.M., Kelton, W.D., Kelton, W.D.: Simulation modeling and analysis, vol. 2. McGraw-Hill (1991)

[17] Ming, Z., Luo, C., Gao, W., et al.: Bdgs: A scalable big data generator suite in big data benchmarking. CoRR abs/1401.5465 (2014)

[18] Muñoz, L., Mazón, J.N., Pardillo, J., Trujillo, J.: Modelling etl processes of data warehouses with uml activity diagrams. In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM Workshops. Lecture Notes in Computer Science, vol. 5333, pp. 44–53. Springer (2008)

[19] Pall, A.S., Khaira, J.S.: A comparative review of extraction, transformation and loading tools. Database Systems Journal 4(2), 42–51 (2013)

[20] Paul, R.J., Hlupic, V., Giaglis, G.M.: Simulation modelling of business processes. In: Proceedings of the 3 rd U.K. Academy of Information Systems Conference, McGraw-Hill. pp. 311–320. McGraw-Hill (1998)

[21] Rabl, T., Frank, M., Sergieh, H.M., Kosch, H.: A data generator for cloud-scale benchmarking. In: TPCTC. pp. 41–56 (2010)

[22] Simitsis, A., Vassiliadis, P., Dayal, U., Karagiannis, A., Tziovara, V.: Benchmarking etl workflows. In: TPCTC. pp. 199–220 (2009)

[23] Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.: Qox-driven etl design: reducing the cost of etl consulting engagements. In: SIGMOD Conference. pp. 953–960 (2009)

[24] Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.: QoX-driven ETL design: reducing the cost of ETL consulting engagements. In: SIGMOD. pp. 953–960 (2009)

[25] Strange, K.H.: ETL Was the Key to This Data Warehouse's Success (March 2002), Gartner Research, CS-15-3143

[26] Theodorou, V., Abelló, A., Lehner, W.: Quality Measures for ETL Processes. In: DaWaK. pp. 9–22 (2014)

[27] Vassiliadis, P., Simitsis, A., Baikousi, E.: A taxonomy of etl activities. In: Song, I.Y., Zimányi, E. (eds.) DOLAP. pp. 25–32. ACM (2009)

[28] Vassiliadis, P., Simitsis, A., Skiadopoulos, S.: Conceptual modeling for ETL processes. In: DOLAP. pp. 14–21 (2002)

[29] Wang, L., Zhan, J., et al.: BigDataBench: a Big Data Benchmark Suite from Internet Services. CoRR abs/1401.1406 (2014)

[30] Wilkinson, K., Simitsis, A., Castellanos, M., Dayal, U.: Leveraging business process models for etl design. In: ER. pp. 15–30 (2010)

[31] Zhang, J., Xu, C., Cheung, S.C.: Automatic generation of database instances for white-box testing. In: COMPSAC. pp. 161–165 (2001)