Bijoux: Data Generator for Evaluating ETL Process Quality

Vasileios Theodorou[a], Petar Jovanovic[a], Alberto Abelló[a], Emona Nakuçi[a]

[a] *Universitat Politècnica de Catalunya, BarcelonaTech*
Barcelona, Spain ({vasileios|petar|aabello}@essi.upc.edu), emona.nakuci@est.fib.upc.edu

Abstract

Obtaining the right set of data for evaluating the fulfillment of different quality factors 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 manually providing a synthetic set of data is known as a labor-intensive task that needs to take various combinations of process parameters into account. More importantly, 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 input data, and automatically generates testing datasets. *Bijoux* is highly modular and configurable to enable end-users to generate datasets for a variety of interesting test scenarios (e.g., evaluating specific parts of an input ETL process design, with different input dataset sizes, different distributions of data, and different operation selectivities). We have developed a running prototype that implements the functionality of our data generation framework and here we report our experimental findings showing the effectiveness and scalability of our approach.

Keywords: Data generator, ETL, process quality

1. Introduction

Data-intensive processes constitute a crucial part of complex business intelligence (BI) systems responsible for delivering information to satisfy the needs of different end users. Besides delivering the right information to end users, data-intensive 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; see [1, 2, 3]), an extensive set of experiments over the designed process must be performed to test the behavior of the process in a plethora of possible execution scenarios. Essentially, the properties of input data (e.g., value distribution, cleanness, consistency) play a major role in evaluating the resulting quality characteristics of a data-intensive process. Furthermore, to obtain the finest level of granularity of process metrics, quantitative analysis techniques for business processes (e.g., [4]) propose analyzing the quality characteristics at the level of individual activities and resources. Moreover, one of the most popular techniques for quantitative analysis of process models is process simulation [4], which assumes creating large number of hypothetical process instances that will simulate the execution of the process flow for different scenarios. In the case of data-intensive processes, the simulation should be additionally accompanied by a sample of input data (i.e., *work item* in the language of [4]) created for simulating a specific scenario.

Nonetheless, obtaining input data for performing such experiments is rather challenging. Sometimes, easy access to the real source data is hard, either due to data confidentiality or high data transfer costs. However, in most cases the complexity comes from the fact that a single instance of available data, 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., [5]) have tackled the problem of synthetic test data generation. However, the main focus was on testing the correctness of the developed systems, rather than evaluating different data quality characteristics, which are critical when designing data-intensive processes. Moreover, since the execution of data-intensive processes is typically fully automated and time-critical, ensuring their correct, efficient and reliable execution, as well as certain levels of data quality of their produced output is pivotal.

In the data warehousing (DW) context, an example of a complex, data intensive and often error-prone data-intensive process is the extract-transform-load (ETL) process, responsible for periodically populating a data warehouse from the available data sources. Gartner has reported in [6] that the correct ETL implementation may take up to 80% of the entire DW project. Moreover, the ETL design tools available in the market [7] 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 evaluating 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 different quality characteristics of the process design. To this end, we propose an automated data generation framework for evaluating ETL processes (i.e., *Bijoux*). Growing amounts of data represent hidden treasury assets of an enterprise. However, due to dynamic business environments, data quickly and unpredictably evolve, possibly making the software that processes them (e.g., ETL) inefficient and obsolete. Therefore, we need to generate delicately crafted sets of data (i.e., *bijoux*) to test different execution scenarios of an ETL process and detect its behavior (e.g., *performance*) over a variety of changing parameters (e.g., *dataset size*, *process complexity*, *input data quality*) .

For overcoming the complexity and heterogeneity of typical ETL processes, we tackle the problem of formalizing the semantics of ETL operations and classifying the operations based on the part of input data they access for processing. This largely facilitates *Bijoux* during data generation processes both for identifying the constraints that specific operation semantics imply over input data, as well as for deciding at which level the data should be generated (e.g., single field, single tuple, complete dataset).

Furthermore, *Bijoux* offers data generation capabilities in a modular and configurable manner. Instead of relying on the default data generation functionality provided by the tool, more experienced users may also select specific parts of an input ETL process, as well as desired quality characteristics to be evaluated using generated datasets.

To illustrate the functionality of our data generation framework, we introduce the running toy example that shows an ETL process (see Figure 1), which is a simplified implementation of the process defined in the *TPC-DI benchmark*[1] for loading the *DimSecurity* table during the *Historical Load* phase[2]. The ETL process extracts data from a file with fixed-

[1] `http://www.tpc.org/tpcdi/`

[2] Full implementation available at: `https://github.com/AKartashoff/TPCDI-PDI/`

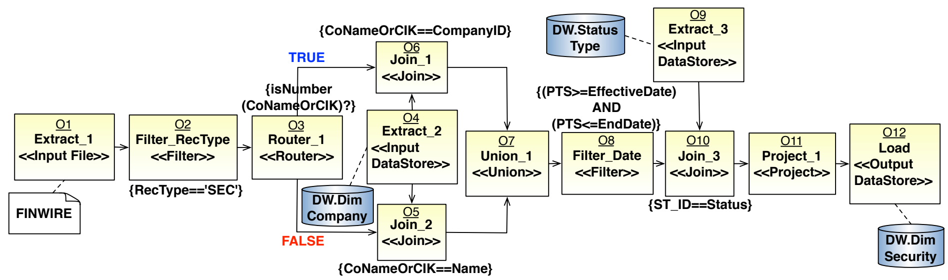


Figure 1: ETL flow example: TPC-DI DimSecurity population

width fields (flat file in the *Staging Area*), which is a merged collection of financial information about companies and securities coming from a financial newswire (FINWIRE) service. The input set is filtered to keep only records about Securities (RecType==‘SEC’) and then rows are split to two different routes, based on whether or not their values for the field *CoNameOrCIK* are numbers (isNumber(CoNameOrCIK)) or not. For the first case, data are matched with data about companies through an equi-join on the company ID number (CoNameOrCIK==CompanyID). On the other hand, for the second case, data are matched with data about companies through an equi-join on the company name (CoNameOrCIK==Name). In both cases, data about companies are extracted from the *DimCompany* table of the data warehouse. Subsequently, after both routes are merged, data are filtered to keep only records for which the posting date and time (PTS) correspond to company data that are current ((PTS>=EffectiveDate) AND (PTS<=EndDate)). Lastly, after data are matched with an equi-join to the data from the StatusType table, to get the corresponding status type for each status id (ST_ID==Status), only the fields of interest are maintained through a projection and then data are loaded to the *DimSecurity* table of the DW.

For the sake of simplicity, in what follows we will refer to the operators of our example ETL, using the label noted for each operator in Figure 1 (i.e., `O1` for `Extract_1`, `O2` for `Filter_RecType`, etc.). Given that an ETL process model can be seen as a directed acyclic graph (DAG), *Bijoux* follows a topological order of its nodes, i.e., operations (e.g., `O1`, `O2`, `O3`, `O4`, `O5`, `O6`, `O7`, `O8`, `O9`, `10`, `O11`,and `O12`), and extracts the found flow constraints (e.g., RecType==‘SEC’ or CoNameOrCIK==Name). Finally, *Bijoux* generates the data that satisfy the given constraints and can be used to simulate the execution of the given ETL process.

Our framework, *Bijoux*, is useful during the early phases of the ETL process 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 (e.g., *adding new data sources*, *adding mechanisms for improving data consistency*). Finally, the *Bijoux*'s functionality for automated generation of synthetic data is also relevant during the ETL process deployment. It provides users with the valuable benchmarking support (i.e., synthetic datasets) when selecting 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 framework and its algorithms for the automatic data generation. Section 4 introduces modified versions of our example ETL process and showcases the benefits of *Bijoux* for re-evaluating flow changes. In Section 5, we introduce the architecture of the prototype system that implements the functionality of the *Bijoux* framework and further report our experimental results. Finally, Section 6 discusses the related work, while Section 7 concludes the paper and discusses possible future directions.

2. Overview of our approach

In this section, we present the overview of our data generation framework. We classify the ETL process operations and formalize the ETL process elements in the context of data generation and subsequently, in a nutshell, we present the overview of the data generation process of the *Bijoux* framework.

2.1. ETL operation classification

To ensure applicability of our approach to ETL processes coming from major ETL design tools and their typical operations, we performed a comparative study 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 noticed that some of these tools have a very broad palette of specific operations (e.g., PDI has a support for invoking external web services for

Table 1: Comparison of ETL operations through selected ETL tools - Part 1

Operation Level	Operation Type	Pentaho PDI	Talend Data Integration	SSIS	Oracle Warehouse Builder
Field	Field Value Alteration	Add constant Formula Number ranges Add sequence Calculator Add a checksum	tMap tConvertType tReplaceList	Character Map Derived Column Copy Column Data Conversion	Constant Operator Expression Operator Data Generator Transformation Mapping Sequence
Dataset	Duplicate Removal	Unique Rows Unique Rows (HashSet)	tUniqRow	Fuzzy Grouping	Deduplicator
	Sort	Sort Rows	tSortRow	Sort	Sorter
	Sampling	Reservoir Sampling Sample Rows	tSampleRow	Percentage Sampling Row Sampling	
	Aggregation	Group by Memory Group by	tAggregateRow tAggregateSortedRow	Aggregate	Aggregator
	Dataset Copy		tReplicate	Multicast	
Row	Duplicate Row	Clone Row	tRowGenerator		
	Filter	Filter Rows Data Validator	tFilterRow tMap tSchemaComplianceCheck	Conditional Split	Filter
	Join	Merge Join Stream Lookup Database lookup Merge Rows Multiway Merge Join Fuzzy Match	tJoin tFuzzyMatch	Merge Join Fuzzy Lookup	Joiner Key Lookup Operator
	Router	Switch/Case	tMap	Conditional Split	Splitter
	Set Operation - Intersect	Merge Rows (diff)	tMap	Merge Join	Set Operation
	Set Operation - Difference	Merge Rows (diff)	tMap		Set Operation
	Set Operation - Union	Sorted MergeAppend streams	tUnite	Merge Union All	Set Operation

Table 2: Comparison of ETL operations through selected ETL tools - Part 2

Operation Level	Operation Type	Pentaho PDI	Talend Data Integration	SSIS	Oracle Warehouse Builder
Schema	Field Addition	Set field value Set field value to a constant String operations Strings cut Replace in string Formula Split Fields Concat Fields Add value fields changing sequence Sample rows	tMap tExtractRegexFields tAddCRCRow	Derived Column Character Map Row Count Audit Transformation	Constant Operator Expression Operator Data Generator Mapping Input/Output parameter
	Datatype Conversion	Select Values	tConvertType	Data Conversion	Anydata Cast Operator
	Field Renaming	Select Values	tMap	Derived Column	
	Projection	Select Values	tFilterColumns		
Table	Pivoting	Row Denormalizer	tDenormalize tDenormalizeSortedRow	Pivot	Unpivot
	Unpivoting	Row Normalizer Split field to rows	tNormalize tSplitRow	Unpivot	Pivot
Value	Single Value Alteration	If field value is null Null if Modified Java Script Value SQL Execute	tMap tReplace	Derived Column	Constant Operator Expression Operator Match-Merge Operator Mapping Input/Output parameter
Source Operation	Extraction	CSV file input Microsoft Excel Input Table input Text file input XML Input	tFileInputDelimited tDBInput tFileInputExcel	ADO .NET / DataReader Source Excel Source Flat File Source OLE DB Source XML Source	Table Operator Flat File Operator Dimension Operator Cube Operator
Target Operation	Loading	Text file output Microsoft Excel Output Table output Text file output XML Output	tFileOutpu tDelimited tDBOutput tFileOutputExcel	Dimension Processing Excel Destination Flat File Destination OLE DB Destination SQL Server Destination	Table Operator Flat File Operator Dimension Operator Cube Operator

Table 3: List of operations considered in the framework

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

performing the computations specified by these services). Moreover, some operations can be parametrized to perform different kinds of transformation (e.g., tMap in Talend), while others can have overlapping functionalities, or different implementations for the same functionality (e.g., *FilterRows* and *JavaFilter* in PDI). Tables 1 and 2 show the resulting classification of the ETL operations from the considered tools.

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 3). Notice that this list covers all operations of our running example in Figure 1, except extraction and loading ones, which are not assumed to carry any specific semantics over input data and thus are not considered operations by our classification.

A similar study of typical ETL operations inside several ETL tools has been performed before in [8]. 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, we classify ETL operations based on the part of the input table they access when processing the data (i.e., *table*, *dataset*, *row*, *schema*, *field*, or *field value*; see the first column of Table 1 and Table 2) in order to assist *Bijoux* when deciding at which level data should be generated. In Figure 2, we conceptually depict the relationships between different parts of input

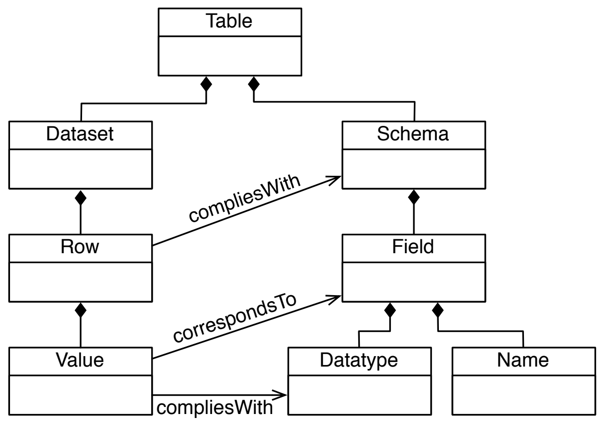


Figure 2: Table-access based classification, UML notation

data, which forms the basis for our ETL operation classification. In our approach, we consider the *Name* of a *Field* to act as its identifier.

2.2. Formalizing ETL processes

The modeling and design of ETL processes is a thoroughly studied area, both in the academia [9, 10, 11, 12] and industry, where many tools available in the market often provide overlapping functionalities for the design and execution of ETL processes [7]. 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 a logical (platform-independent) representation of an ETL process, which in the literature is usually represented as a directed acyclic graph (DAG) [12, 13]. We thus formalize an ETL process as a DAG consisting of a set of nodes ($\mathbf{V}$), which are either source or target data stores ($\mathbf{DS} = \mathbf{DS}_S \cup \mathbf{DS}_T$) 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 graph are defined by a schema (i.e., finite list of fields) and a connection to a source ($\mathbf{DS}_S$) or a target ($\mathbf{DS}_T$) 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. Notice that we model input and output data of an ETL process in terms of one or more *tables* (see Figure 2).

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

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

- $\mathbb{I} = \{I_1, \ldots, I_n\}$ is a finite set of input tables.
- $\mathbb{O} = \{O_1, \ldots, O_m\}$ is a finite set of output tables.
- $\mathbf{X}$ ($\mathbf{X} \subseteq Attr(\mathbb{I})$) is a subset of fields of the input tables $\mathbb{I}$ required by the operation. Notice that the function *Attr* for a given set of input or output tables, returns a set of fields (i.e., attributes) that builds the schema of these tables.
- $\mathbf{S} = (\mathbb{P}, \mathbb{F})$ represents ETL operation semantics in terms of:
  - $\mathbb{P} = \{P_1(X_1), \ldots P_p(X_p)\}$: a set of conjunctive predicates over subsets of fields in $\mathbf{X}$ (e.g., $Age > 25$).
  - $\mathbb{F} = \{F_1(X_1), \ldots F_f(X_f)\}$: a set of functions applied over subsets of fields in $\mathbf{X}$ (e.g., $Substr(Name, 0, 1)$). The results of these functions are used either to alter the existing fields or to generate new fields in the output table.
- $\mathbf{A}$ is the subset of fields from the output tables, added or altered during the operation.

Intuitively, the above ETL notation defines a transformation of the input tables ($\mathbb{I}$) into the result tables ($\mathbb{O}$) by evaluating the predicate(s) and function(s) of semantics $\mathbf{S}$ over the functionality schema $\mathbf{X}$ and potentially generating or altering fields in $\mathbf{A}$.

An ETL operation processes input tables $\mathbb{I}$, 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 *row* (i.e., passed/modified/generated/removed rows), and (3) the *dataset* level (i.e., output cardinality).

In Table 4, we formalize the semantics of ETL operations considered by the framework (i.e., operations previously listed in Table 3). Notice that some operations are missing from Table 4, as they can be derived

Table 4: Table of ETL operations semantics

Op. Level	Op. Type	Op. Semantics
Value	Single Value Alteration	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{I}) = Attr(\mathbb{O}) \wedge \lvert\mathbb{I}\rvert = \lvert\mathbb{O}\rvert))$ $\forall t_{in} \in \mathbb{I}(P_i(t_{in}[\mathbf{X}]) \rightarrow \exists t_{out} \in \mathbb{O}(t_{out}[Attr(\mathbb{O}) \setminus \mathbf{A}] = t_{in}[Attr(\mathbb{I}) \setminus \mathbf{A}] \wedge t_{out}(A) = F_j(t_{in}[\mathbf{X}])))$
Field	Field Alteration	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{I}) = Attr(\mathbb{O}) \wedge \lvert\mathbb{I}\rvert = \lvert\mathbb{O}\rvert))$ $\forall t_{in} \in \mathbb{I},\ \exists t_{out} \in \mathbb{O}(t_{out}[Attr(\mathbb{O}) \setminus A] = t_{in}[Attr(\mathbb{I}) \setminus A] \wedge t_{out}(A) = F_j(t_{in}[\mathbf{X}])))$
Row	Duplicate Row	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{I}) = Attr(\mathbb{O}) \wedge \lvert\mathbb{I}\rvert < \lvert\mathbb{O}\rvert))$ $\forall t_{in} \in \mathbb{I}, \exists \mathbb{O}' \subseteq \mathbb{O}, \lvert\mathbb{O}'\rvert = n\ ^3\ \wedge \forall t_{out} \in \mathbb{O}', t_{out} = t_{in}$
	Router	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow \forall j(Attr(O_j) = Attr(\mathbb{I}) \wedge \lvert\mathbb{I}\rvert \geq \lvert O_j\rvert))$ $\forall j, \forall t_{in} \in \mathbb{I}(P_j(t_{in}[x_j]) \rightarrow \exists t_{out} \in O_j, (t_{out} = t_{in})$
	Filter	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{O}) = Attr(\mathbb{I}) \wedge \lvert\mathbb{I}\rvert \geq \lvert\mathbb{O}\rvert))$ $\forall t_{in} \in \mathbb{I}(P_j(t_{in}[\mathbf{X}]) \rightarrow \exists t_{out} \in \mathbb{O}, (t_{out} = t_{in})$
	Join	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{O}) = Attr(I_1) \cup Attr(I_2) \wedge \lvert\mathbb{O}\rvert \leq \lvert I_1 \times I_2\rvert))$ $\forall t_{in_1} \in I_1, t_{in_2} \in I_2, (P(t_{in_1}[x_1], t_{in_2}[x_2]) \rightarrow \exists t_{out} \in \mathbb{O}(t_{out} = t_{in_1} \bullet t_{in_1})$
	Union	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(I_1) = Attr(I_2) \wedge Attr(\mathbb{O}) = Attr(I_1) \wedge \lvert\mathbb{O}\rvert = \lvert I_1\rvert + \lvert I_2\rvert))$ $\forall t_{in} \in (I_1 \cup I_2) \rightarrow \exists t_{out} \in \mathbb{O}(t_{out} = t_{in})$
	Difference	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(I_1) = Attr(I_2) \wedge Attr(\mathbb{O}) = Attr(I_1) \wedge \lvert\mathbb{O}\rvert \leq \lvert I_1\rvert))$ $\forall t_{in}(t_{in} \in I_1 \wedge t_{in} \notin I_2) \rightarrow \exists t_{out} \in \mathbb{O}(t_{out} = t_{in})$
Dataset	Aggregation	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{O}) = \mathbf{X} \cup \mathbf{A} \wedge Attr(\mathbb{O}) \leq Attr(I)))$ $\forall \mathbb{I}' \in 2^{\mathbb{I}}(\forall t_{in_1} \in \mathbb{I}'(\forall t_{in_2} \in \mathbb{I}'(t_{in_1}[\mathbf{X}] = t_{in_2}[\mathbf{X}]) \wedge \forall t_{in_k} \in \mathbb{I} \setminus \mathbb{I}', t_{in_1}[X] \neq t_{in_k}[X])) \rightarrow$ $\rightarrow \exists! t_{out} \in \mathbb{O}(t_{out}[\mathbf{X}] = t_{in_1}[\mathbf{X}] \wedge t_{out}[\mathbf{A}] = F_j(\mathbb{I}'))$
	Sort	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{I}) = Attr(\mathbb{O}) \wedge \lvert\mathbb{I}\rvert = \lvert\mathbb{O}\rvert))$ $\forall t_{in} \in \mathbb{I}, \exists t_{out} \in \mathbb{O}(t_{out} = t_{in})$ $\forall t_{out}, t_{out'} \in \mathbb{O}(t_{out}[\mathbf{X}] < t_{out'}[\mathbf{X}] \rightarrow t_{out} \prec t_{out'})$
	Duplicate Removal	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{I}) = Attr(\mathbb{O}) \wedge \lvert\mathbb{I}\rvert \geq \lvert\mathbb{O}\rvert))$ $\forall t_{in} \in \mathbb{I}, \exists! t_{out} \in \mathbb{O}(t_{out} = t_{in})$
	Dataset Copy	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow \forall j(Attr(O_j) = Attr(\mathbb{I}) \wedge \lvert\mathbb{I}\rvert = \lvert O_j\rvert))$ $\forall j, \forall t_{in} \in \mathbb{I},\ \exists t_{out} \in O_j, (t_{out} = t_{in})$
Schema	Projection	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{O}) = Attr(\mathbb{I}) \setminus \mathbf{X} \wedge \lvert\mathbb{I}\rvert = \lvert\mathbb{O}\rvert))$ $\forall t_{in} \in \mathbb{I}, \exists t_{out} \in \mathbb{O}(t_{out}[Attr(\mathbb{O})] = t_{in}[Attr(\mathbb{I}) \setminus \mathbf{X}]))$
	Field Renaming	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{O}) = (Attr(\mathbb{I}) \setminus \mathbf{X}) \cup \mathbf{A}) \wedge \lvert\mathbb{I}\rvert = \lvert\mathbb{O}\rvert))$
	Field Addition	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{O}) = Attr(\mathbb{I}) \cup \mathbf{A} \wedge \lvert\mathbb{I}\rvert = \lvert\mathbb{O}\rvert))$ $\forall t_{in} \in \mathbb{I}, \exists t_{out} \in \mathbb{O}(t_{out}[Attr(\mathbb{O}) \setminus \mathbf{A}] = t_{in}[Attr(\mathbb{I})] \wedge t_{out}[\mathbf{A}] = F(t_{in}[\mathbf{X}]))$
Table	Pivoting	$\forall(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A})(F(\mathbb{I},\mathbb{O},\mathbf{X},\mathbf{S},\mathbf{A}) \rightarrow (Attr(\mathbb{O}) = (Attr(\mathbb{I}) \setminus \mathbf{X}) \cup \mathbf{A} \wedge \lvert\mathbb{O}\rvert = \lvert\mathbb{I}\rvert_a \wedge \lvert\mathbb{I}\rvert = \lvert\mathbb{O}\rvert_a))$ $\forall t_{in} \in \mathbb{I}, \forall a \in Attr(\mathbb{I}), \exists t_{out} \in \mathbb{O}, \exists b \in Attr(\mathbb{O})(t_{out}[b] = t_{in}[a]))$

from the semantics of other listed operations (e.g., Intersection as a special case of Join, Unpivoting as an inverse operation to Pivoting, and Datatype Conversion as a special case of Field Alteration using a specific conversion function).

In our approach, we use such formalization of operation semantics to automatically extract the constraints that an operation implies over the input data, hence to further generate the input data for covering such operations. However, notice that some operations in Table 4 may imply specific semantics over input data that are not explicitly expressed in the given formalizations (e.g., *Field Addition/Alteration*, *Single Value Alteration*). Such semantics may span from simple arithmetic expressions (e.g., $yield = divident \div DM\_CLOSE$), to complex user defined functions expressed in terms of an ad hoc script or code snippets. While the former case can be easily tackled by powerful expression parsers [13], in the later case the operation's semantics must be carefully analyzed to extract the constraints implied over input data (e.g., by means of the static code analysis, as suggested in [14]).

2.3. Bijoux overview

Intuitively, starting from a logical model of an ETL process and the semantics of ETL operations, *Bijoux* analyzes how the fields 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 fields. 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 fields (i.e., *value ranges*, *datatypes*, and *the sizes of generated data*). Using the collected parameters, *Bijoux* then generates input datasets to satisfy all found constrains, i.e., to simulate the execution of selected parts of the 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*), fields (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. Bijoux data generation framework

The data generation process includes four main stages (i.e., 1 - *path enumeration*, 2 - *constraints extraction*, 3 - *constraints analysis*, and 4 - *data generation*).

3.1. Preliminaries and Challenges

We first discuss some of the important challenges of generating data for evaluating general ETL flows, as well as the main structures maintained during the data generation process.

The workflow-graph structure of the ETL logical model that we adopt for our analysis consists of ETL operations as graph *nodes*, input data stores as graph *sources* and output data stores as graph *sinks*. In particular, input

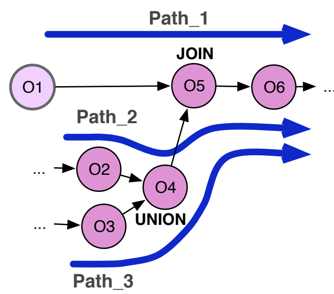
[3] n is the number of replicas in the Replicate Row operation semantics

data stores, as well as routing operations (e.g., Routers) that direct rows to different outputs based on specified conditions, introduce alternative directed paths of the input graph (in the rest of the paper referred to as *paths*), which can be followed by input data. Hence, there are two properties of the generated input data that can be defined:

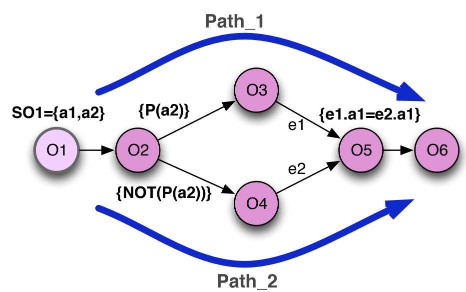
- *Path Coverage*: Input data are sufficient to "cover" a specific path, i.e., each and every edge (or node) that is on this path is visited by at least one row of data.

- *Flow Coverage*: Input data are sufficient to "cover" the complete flow graph, i.e., each and every edge (or node) of the flow graph is visited by at least one row of data.

The apparently simple case of *Path Coverage* hides an inherent complexity, deriving from the fact that some joining operations (i.e., *joining nodes*; e.g., Join, Intersection) require the involvement of multiple paths in order to direct data to their output. In addition, new fields are introduced to the flow either through input data stores or Field Addition operations (see Table 4), while the fields from different paths are *fused*/joined together through joining operations. This in turn implies two facts: i) *Path Coverage* is not guaranteed by generating the right input data only for the input data store that is involved in a specific path; instead, data generation should be conducted for a combination of paths (i.e., their included input data stores), and ii) during the *Path Coverage* analysis, referring to a field solely by its name is not sufficient; the same field might participate in multiple paths from a combination of paths, in each path holding different properties coming from extracted constraints of different operations. Thus, the *name* of a field should be combined with a *pathid* to identify one distinct entity with specific properties.



(a) Alternative path combinations for coverage of the same path



(b) Multiple rows from same input source required for coverage

Figure 3: Notable cases of graph patterns

In Figure 3, we show some notable cases of graph patterns that require special attention during the coverage analysis, as described above.

In Figure 3a, we can see how the coverage of `Path_1` (O1→O5→O6...) needs multiple paths to be considered for data generation, because of the joining operation O5 that requires multiple inputs (e.g., a *Join* operation). Thus, coverage can be ensured by using alternative combinations, either `Path_1` in combination with `Path_2` (...O2→O4→O5→O6...), or `Path_1` in combination with `Path_3` (...O2→O4→O5→O6...). It should be mentioned that operation O4 is of a merging type that does not require both of its incoming edges to be crossed in order to pass data to its output (i.e., a *Union* operation) and thus `Path_2` and `Path_3` can be used interchangeably for coverage.

In Figure 3b, we show how the coverage of one path might require the generation of multiple rows for the same input source. For example, for the Path Coverage of `Path_1` (O1→O2→O3→O5→O6) it is required to additionally generate data for `Path_2` (O1→O2→O4→O5→O6), because of the existence of the joining operation O5. It should be noticed here that fields *a1* and *a2* in `Path_1` belong to a different instance than in `Path_2`, since the condition of the routing operator O2 imposes different predicates over *a2* for different paths (i.e., P(a2) and NOT(P(a2)), respectively). Hence, at least two different rows from the same input data store are required for *Path Coverage* of `Path_1`.

Example. For illustrating the functionality of our algorithm, we will use the running example introduced in Section 1 (see Figure 1). For the sake of simplicity, we will not use the complete schemata of the input data stores as specified in the *TPC-DI benchmark*, but instead we assume simplified versions, where the only fields present are the ones that are used in the ETL flow, i.e., taking part in predicates or functions. In this manner, input data stores of the example ETL flow are: $\mathbb{I}$ = {O1, O4, O9}, with schemata SO1 = {PTS, RecType, Status, CoNameOrCIK}, SO4 = {CompanyID, Name, EffectiveDate, EndDate} and SO9 = {ST_ID, ST_NAME}; whilst a topological order of its nodes is: {O1, O2, O3, O4, O5, O6, O7, O8, O9, O10, O11, O12}. Besides this running example, we will also use the auxiliary example graph from Figure 4a to support the description of the complete functionality of *Bijoux* □

3.2. Data structures

Before going into the details of algorithms 1 and 2 in Section 3.4, we present the main structures maintained by these algorithms.

While analyzing a given ETL graph, in Algorithm 1, *Bijoux* builds the following structures that partially or completely record the path structures of the input ETL graph (i.e., path traces):

- *Path Traces* ($\mathbb{PT}$) collection keeps traces of operations and edges that have been visited, when following a specific path up to a specific node in the ETL graph. Traces of individual paths PT ($PT \in \mathbb{PT}$) are built incrementally and thus, following a specific path on the graph, if a Path Trace PT1 is generated at an earlier point than the generation of a Path Trace PT2, then PT1 will include a subset of the trace of PT2 (i.e., $PT1 \subseteq PT2$). From an implementation point of view, each PT holds a *Signature* as a property, which can be a string concatenation of graph elements that shows which route has been followed in the case of alternative paths. This enables very efficient PT analysis and comparisons by simply applying string operations.

  Example. Referring to our running example in Section 1 we can have the following signature of a Path Trace PT1:
  $Sig(PT1) =$ "$I[O1].S[O2, true].S[O3, true].J[O6, e1]$"

From this signature we can conclude that PT1 starts from I (i.e., Input Source): *O1*; passes through S (i.e., Splitting Operation): *O2* coming from its outgoing edge that corresponds to the evaluation: *true* of its condition; passes through S (i.e., Splitting Operation): *O3* coming from its outgoing edge that corresponds to the evaluation: *true*; passes through J (i.e., Joining Operation): *O6* coming from its incoming edge: *e1*; and so on. For some operations (e.g., Joins) it makes sense to keep track of the incoming edge through which they have been reached in the specific path and for some others (e.g., Routers), it makes sense to keep track of the outgoing edge that was followed for the path.
Looking at the following signature of Path Trace PT2:
$Sig(PT2)$ = "$I[O1].S[O2, true].S[O3, true]$" , we can infer that PT1 and PT2 are on the same path of the ETL graph, PT2 being generated at an "earlier" point, since the signature of PT2 is a substring of the signature of PT1. □

- *Tagged Nodes* ($\mathbb{TN}$) structure records, for each node, the set of paths (i.e., operations and edges) reaching that node from the input data store nodes (i.e., source nodes). Thus, each node is "tagged" with a set of Path Traces ($\mathbb{PT}$) which are being built incrementally, as explained above.
  Example. Referring to our running example, within $\mathbb{TN}$ the *O7* operation node will be "tagged" with four different path traces, $PT1$, $PT2$, $PT3$ and $PT4$ with the following signatures:
  - $Sig(PT1)$ = "$I[O1].S[O2, true].S[O3, true].J[O6, e1].J[O7, e1]$"
  - $Sig(PT2)$ = "$I[O1].S[O2, true].S[O3, false].J[O5, e1].J[O7, e2]$"
  - $Sig(PT3)$ = "$I[O4].J[O6, e2].J[O7, e1]$"
  - $Sig(PT4)$ = "$I[O4].J[O5, e2].J[O7, e2]$" □

- *Final path traces* ($\mathbb{FP}$) structure records all the *complete* (i.e., *source-to-sink*) paths from the input ETL graph, by maintaining all source-to-sink Path Traces (i.e., the union of all Path Traces that tag sink nodes).

When it comes to formalizing the main structure that is being built by Algorithm 2 (i.e., data generation pattern), we define its structure as follows:

- A data generation pattern ($Pattern$) consists of a set of path constraints (i.e., $pathConstr$), where each path constraint is a set of constraints over the input fields introduced by the operations of an individual path. Formally:

$Pattern = \{pathConstr_i | i = 1, \cdots, pathNum\}$

Example. In our running example (Figure 1), so as to cover the path Path1=(O1→O2→O3→O6→O7→O8→O10→O11→O12), additionally, the path Path2=(O4→O6→O7→O8→O10→O11→O12) and the path Path3=(O9→O10→O11→O12) need to be covered as well, because of the equi-join operators *O6* and *O10*. The *Pattern* would then consist of three constraints sets ($pathConstr1$, $pathConstr2$ and $pathConstr3$), one for each (source-to-sink) path of the flow that has to be covered. □

- A path constraint (i.e., $pathConstr_i$) consists of a set of constraints over individual fields of the given path (i.e., $fieldConstr$). Formally: $pathConstr_i = \{fieldConstr_j | j = 1, \cdots, pathFieldNum\}$
  Example. Each constraints set in our example will contain a set of constraints for any of the fields that are involved in imposed predicates of operations on the related path. For example, $pathConstr1$ will contain constraints over the fields: Path1.PTS, Path1.RecType, Path1.Status, Path1.CoNameOrCIK, Path1.CompanyID, Path1.Name, Path1.EffectiveDate, Path1.EndDate, Path1.ST_ID, Path1.ST_name. Notice that each field is also defined by the related path. Respectively, $pathConstr2$ and $pathConstr3$ will contain constraints over the same fields as $pathConstr1$, but with the corresponding path as identifier (e.g., Path2.PTS, Path2.RecType and so on for $pathConstr2$ and Path3.PTS, Path3.RecType and so on for $pathConstr3$). In our example, it does not make any difference maintaining constraints coming from fields of O4 for Path1 (for e.g., CompanyId for Path1), since the flow is not split after it merges, but in the general case they are necessary for cases of indirect implications over fields from one path and for determining the number of rows that need to be generated. □

- A field constraint (i.e., $fieldConstr_j$) is defined as a pair of an input field and an ordered list of constraint predicates over this field. Formally:
  $fieldConstr_j = [field_j, \mathbb{S}_j]$
  Example. An example field constraint that can be found in our running scenario within $pathConstr1$, is:
  $fieldConstr_1 = [Path1.RecType, \{(RecType == `SEC')\}]$ □

- Finally, a constraint predicates list defines the logical predicates over the given field in the topological order they are applied over the field in the given path. Formally:

$\mathbb{S}_j =< P_1(field_j), \cdots, P_{constrNum}(field_j) >$

The list needs to be ordered to respect the order of operations, since in the general case:

$f_1(f_2(field_x)) \neq f_2(f_1(field_x))$

After processing the input ETL graph in Algorithm 1, Algorithm 2 uses the previously generated collection of final path traces (i.e., $\mathbb{FP}$) for traversing a selected *complete* path (i.e., $PT \in \mathbb{FP}$) and constructing a *data generation pattern* used finally for generating data that will guarantee its *coverage*. Thus, Algorithm 2 implements the construction of a data generation pattern for *path coverage* of one specific path. For *flow coverage* we can repeat Algorithm 2, starting every time with a different PT from the set of final path traces $\mathbb{FP}$, until each node of the ETL graph has been visited at least once. We should notice here that an alternative to presenting two algorithms — one for path enumeration and one for pattern construction — would be to present a merged algorithm, which traverses the ETL graph and at the same time extracts constraints and constructs the data generation pattern. However, we decided to keep Algorithm 1 seperate for two reasons: i) this way the space complexity is reduced while computational complexity remains the same and ii) we believe that the path enumeration algorithm extends beyond the scope of ETL flows and can be reused in a general case for implementing a directed path enumeration in polynomial time, while constructing efficient structures for comparison and analysis (i.e., *Path Traces*). A similar approach of using a compact and efficient way to represent ETL workflows using string signatures has been previously introduced in [15].

3.3. Path Enumeration Stage

In what follows, we present the path enumeration stage, carried out by Algorithm 1.

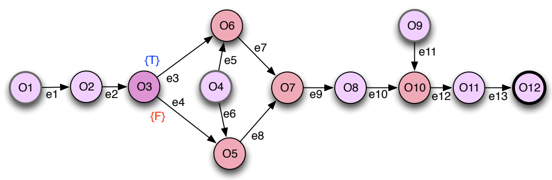
In the initial stage of our data generation process, *Bijoux* processes the input ETL process graph in a topological order (Step 2) and for each source node starts a new path trace (Step 5), initialized with the operation represented by a given source node. At the same time, the source node is tagged by the created path trace (Step 6). For other (non-source) nodes, *Bijoux* gathers the path traces from all the previously tagged predecessor nodes (Step 8), extends these path traces with the current operation o_i (Step 9), while o_i is tagged with these updated path traces ($\mathbb{PT}$). Finally, if the

Algorithm 1 Enumerate Paths and Generate Path Traces

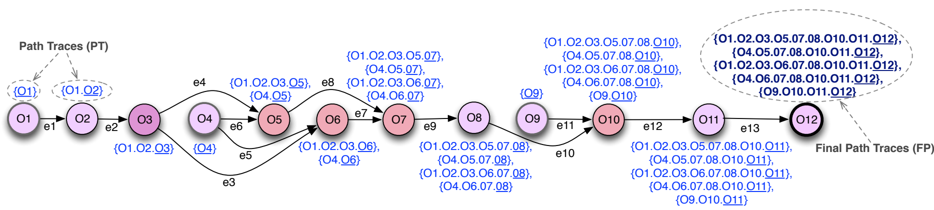
Input: ETL
Output: $\mathbb{FP}$

```
𝕋ℕ ← new Tagged Nodes; 𝔽ℙ ← ∅;
for each operation o_i ∈ TopOrder(ETL) do
    if (o_i is source) then
        ℙ𝕋 ← ∅;
        ℙ𝕋.addElement(new Path Trace(o_i));
        𝕋ℕ.addTag(ℙ𝕋, o_i);
    else
        ℙ𝕋 ← 𝕋ℕ.UnionOfAll_ℙ𝕋s_forAllPredecessorNodesOf(o_i);
        ℙ𝕋.updateBasedOnOperation(o_i);
        if (o_i is sink) then
            𝔽ℙ.addAllElementsFrom(ℙ𝕋);
        else
            𝕋ℕ.addTag(ℙ𝕋, o_i);
        end if
    end if
end for
return 𝔽ℙ;
```

visited operation is a sink node, the traces of the paths that reach this node are added to the list of final path traces (i.e., $\mathbb{FP}$). Processing the input ETL process graph in this manner, Algorithm 1 gathers the complete set of final path traces, that potentially can be covered by the generated input data. An example of the execution of Algorithm 1 applied on our running example and the 5 resulting final path traces are shown in Figure 4.



(a) DAG representation of our running example



(b) Execution of Algorithm 1 for the topological order of the DAG representation of our running example

Figure 4: Example of execution of Algorithm 1

3.4. Constraints Extraction and Analysis Stage

In what follows, we discuss in detail the constraints extraction and analysis stages of our data generation process, carried out by Algorithm 2.

After all possible *final* paths of input ETL graph are processed and their traces recorded in $\mathbb{FP}$, an end-user may select an individual path she wants to *cover*. To this end, *Bijoux* runs Algorithm 2, with a selected path $PT \in \mathbb{FP}$, and builds a data generation *Pattern* to cover (at least) the given path. Algorithm 2 iterates over all the operation nodes of the selected path (Step 2), and for each *joining node* (i.e., node with multiple incoming edges), it searches in $\mathbb{FP}$ for all paths that reach the same *joining node*, from now on, *incident paths* (Steps 5 - 11). As discussed in Section 3.2, routing operations (e.g., *Router*) introduce such paths, and they need to be considered separately when generating data for their coverage (see Figure 3). In general, there may be several *joining nodes* on the selected path, hence Algorithm 2 must take into account all possible combinations of the

Algorithm 2 Construct Data Generation Pattern for one Path

```
Input: ETL, PT, 𝔽ℙ
Output: Pattern
𝔸ℙ ← ∅;
for each operation o_i crossedBy PT do
    if (o_i is of type joining_node) then
        AP_i ← ∅
        for each Path Trace PT_j ∈ 𝕋ℕ.getAllPathTracesFor(o_i) do
            if (PT_j.PredecessorOf(o_i) ≠ PT.PredecessorOf(o_i) ) then
                AP_i.add(PT_j);
            end if
        end for
        𝔸ℙ.add(AP_i);
    end if
end for
ℂ ← allCombinations(PT, 𝔸ℙ);
for each Combination C ∈ ℂ do
    Pattern ← ∅;
    for each Path Trace PT_i ∈ C do
        for each operation o_j crossedBy PT_i do
            Pattern.addConstraints(o_j);
            if (¬Pattern.isFeasible) then
                abortPatternSearchForC();
            end if
        end for
    end for
    return Pattern;
end for
return ∅;
```

alternative incident paths that reach these nodes (Step 13).

Example. Referring to the DAG of Figure 4a, if the path to be covered is (O9→O10→O11→O12), it would require the coverage of additional path(s) because of the equi-join operator *O10*. In other words, data would also need to be coming from edge *e10* in order to be matched with data from edge *e11*. However, because of the existence of a Union operator (*O7*), there are different alternative combinations of paths that can meet this requirement. The reason is that data coming from either of the incoming edges of a *Union* operator reach its outgoing edge. Hence, data reaching *O10* from edge *e10* could pass through path (O1→O2→O3→O6→O7→O8...) combined with path (O4→O6→O7→O8...) or through path (O1→O2→O3→O5→O7→O8...) combined with (O4→O6→O7→O8...). Thus, we see how two alternative combinations of paths, each containing three different paths, can be used for the coverage of one single path. □

For each combination, Algorithm 2 attempts to build a data generation pattern, as explained above. However, some combination of paths may raise a contradiction between the constraints over an input field, which in fact results in disjoint value ranges for this field and thus makes it unfeasible to cover the combination of these paths using a single instance of the input field (Step 20). In such cases, Algorithm 2 aborts pattern creation for a given combination and tries with the next one.

Example. Referring to the DAG of Figure 4a, we can imagine field *f1*, being present in the schema of operation *O6* and field *f2* being present in the schema of operation *O9*. We can also imagine that the datatype of *f1* is *integer* and the datatype of *f2* is *positive integer*. Then, if the joining condition of operation *O10* is ($f1 = f2$) and at the same time, there is a constraint (e.g., in operation *O6*) that ($f1 < 0$), the algorithm will fail to create a feasible data generation pattern for the combination of paths (O1→O2→O3→O5...→O12) and (O9→O10→O11→O12). □

Otherwise, the algorithm updates currently built *Pattern* with the constraints of the next operation (o_j) found on the path trace.

As soon as it finds a combination that does not raise any contradiction and builds a complete feasible *Pattern*, Algorithm 2 finishes and returns the created data generation pattern (Step 24). Notice that by covering at least one combination (i.e., for each *joining node*, each and every incoming edge is crossed by one selected path), Algorithm 2 can guarantee the coverage of the selected input path PT.

Importantly, if Algorithm 2 does not find a feasible data generation pattern for any of the alternative combinations, it returns an empty pattern (Step 26). This further indicates that the input ETL process model is not

	Datatype	Distribution Type	Modification
PTS	Integer	Triangular	-
RecType	String	Uniform Discrete	deform 2%
CoNameOrCIK	String	Complex	-
CompanyID	Long	Uniform	addNullValues 1%
EffectiveDate	Integer	Uniform	setToMaxInt 1%

Field Parameters (FP)

	Selectivity
O2 (Filter_RecType)	0.3
O3 (Router_1)	0.7
O6 (Join_1)	1
O5 (Join_2)	0.95
O8 (Filter_Date)	0.6

Operation Parameters (OP)

Figure 5: Data generation parameters (FP and OP)

correct, i.e., that some of the path branches are not reachable for any combination of input data.

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 at the operation level (OP) (see Figure 5:right). Notice that such parameters can be either obtained by analyzing the input ETL process for a set of previous real executions, or simply provided by the user, for example, for analyzing the flow for a specific set of operation selectivities.

Selectivity of an operation o expresses the ratio of the size of the dataset at the output (i.e., *card(o)*), to the size at the input of an operation (i.e., *input(o)*). Intuitively, for filtering operations, we express selectivity as the percentage of data satisfying the filtering predicate (i.e., $sel(o) = \frac{card(o)}{input(o)}$), while for n-ary (join) operations, for each input e_i, we express it as the percentage of the data coming from this input that will match with other inputs of an operation (i.e., $sel(o, e_i) = \frac{card(o)}{input(o,e_i)}$).

From the *OP* (see Figure 5:right), *Bijoux* finds that operation *O2* (Filter_RecType) has a selectivity of *0.3*. While processing a selected path starting from the operation *O1*, *Bijoux* extracts operation semantics for *O2* and finds that it uses the field *RecType* (*RecType==‘SEC’*). With the selectivity factor of *0.3* from *OP*, *Bijoux* infers that out of all incoming rows for the Filter, 30% should satisfy the constraint that *RecType* should be equal to SEC, while 70% should not. We analyze the selectivity as follows:

- To determine the total number of incoming rows for operation *O8* (Filter_Date), we consider predecessor operations, which in our case come from multiple paths.

- As mentioned above, operation *O2* will allow only 30% of incoming rows to pass. Assuming that the input load size from *FINWIRE* is 1000, this means that in total $0.3 * 1000 = 300$ rows pass the filter condition.

- From these 300 rows only 70%, based on the *O3* (Router_1) selectivity, (i.e., 210 rows) will successfully pass both the filtering (*RecType==‘SEC’*) and the router condition (*isNumber(CoNameOrCIK)*) and hence will be routed to the route that evaluates to $true$. The rest ((i.e., $300 - 210 = 90$ rows)) will be routed to the route that evaluates to $false$.

- The 210 rows that pass both previous conditions, will be matched with rows coming from operation *O4* through the join operation *O6* (Join_1). Since the selectivity of operation *O6* is 1, all 210 tuples will be matched with tuples coming from *O4* and meeting the condition *CoNameOrCIK==CompanyID* and hence will pass the join condition. On the other hand, the selectivity of operation *O5* (Join_2), for the input coming from *O3*(Router_1), is 0.95, which means that from the 90 rows that evaluated to $false$ for the routing condition, only 85 will be matched with tuples coming from *O4* and meeting the condition *CoNameOrCIK==Name*. Thus, $210 + 85 = 295$ tuples will reach the union operation *O6* and pass it.

- Finally, from the 295 rows that will reach operation *O8* (Filter_Date) coming from the preceding union operation, only $0.6 * 295 = 177$ will successfully pass the condition *(PTS>=EffectiveDate) AND (PTS<= EndDate)*, as the selectivity of *OP8* is 0.6.

  In order to generate the data that do not pass a specific operation of the flow, a data generate pattern inverse to the initially generated $Pattern$ in Algorithm 2 needs to be created to guarantee the percentage of data that will fail the given predicate.

Similarly, other parameters can be set for the generated input data to evaluate different quality characteristics of the flow, (see Figure 5:left). As an example, the percentage of null values or incorrect values (e.g., wrong size of telephone numbers or negative age) can be set for the input data, to evaluate the measured data quality of the flow output, regarding *data completeness* and *data accuracy*, respectively. Other quality characteristics like *reliability* and *recoverability* can be examined as well, by adjusting the distribution of input data that result to exceptions and the selectivity of exception handling operations. Examples of the above will be presented in Section 4.

3.5. Data Generation Stage

Lastly, after the previous stage builds data generation patterns for covering either a single path, combination of paths, or a complete flow, the last (data generation) stage proceeds with generating data for each input field f. Data are generated within the ranges (i.e., R) defined by the constraints of the provided pattern, using either random numerical values within the interval or dictionaries for selecting correct values for other (textual) fields.

For each field f, data generation starts from the complete domain of the field's datatype $dt(f)$.

Each constraint P, when applied over the an input field f, generates a set of disjoint ranges of values $R_i^{f,init}$ in which the data should be generated, and each range being inside the domain of the field's datatype $dt(f)$. Formally:

$$P(f) = R^{f,init} = \left\{ r^{f,init} | r^{f,init} \subseteq dt(f) \right\} \tag{1}$$

For example, depending on the field's datatype, a value range for numeric datatypes is an interval of values (i.e., $[x, y]$), while for other (textual) fields it is a set of possible values a field can take (e.g., *personal names*, *geographical names*).

After applying the first constraint P_1, *Bijoux* generates a set of disjoint, non-empty value ranges R_1^f, each range being an intersection with the domain of the field's datatype.

$$\begin{aligned} R_1^f = \Big\{ r_1^f | \forall r_1^{f,init} \in R_1^{f,init}, \exists r_1^f, s.t. : & \qquad (2) \\ (r_1^f = r_1^{f,init} \cap dt(f) \wedge r_1^f \neq \emptyset) \Big\} & \end{aligned}$$

Iteratively, the data generation stage proceeds through all the constraints of the generation pattern. For each constraint P_i it updates the resulting value ranges as an intersection with the ranges produced in the previous step, and produces a new set of ranges R_i^f.

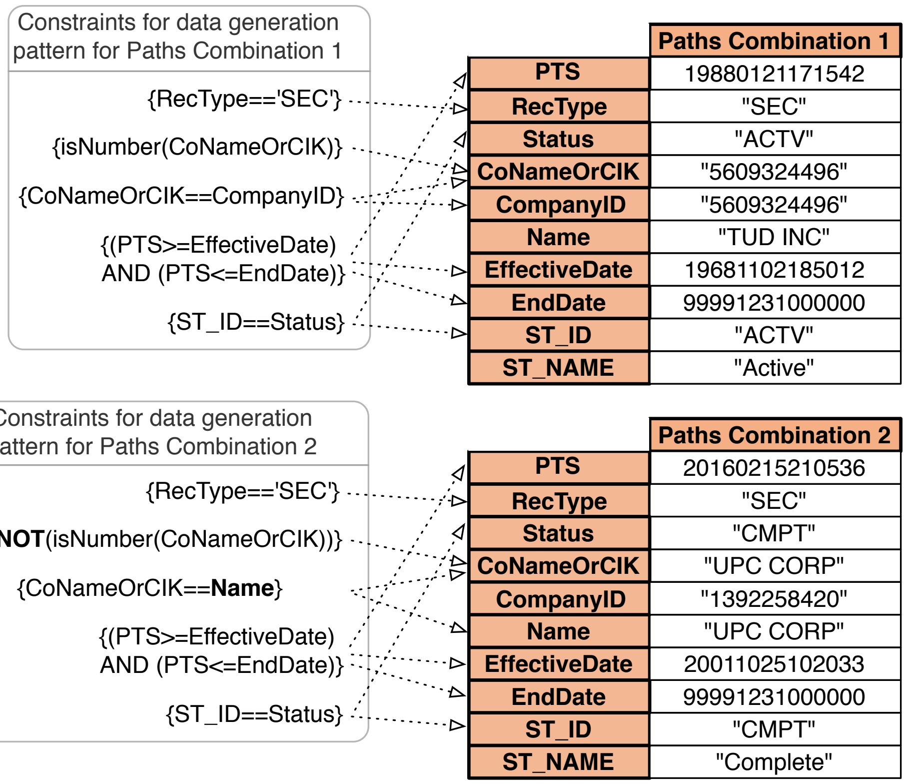


Figure 6: Data generated after analyzing all ETL operations

$$R_i^f = \Big\{ r_i^f | \forall r_i^{f,init} \in R_i^{f,init}, \forall r_{i-1}^f \in R_{i-1}^f, \exists r_i^f, s.t. : \\ (r_i^f = r_i^{f,init} \cap r_{i-1}^f \wedge r_i^f \neq \emptyset) \Big\} \tag{3}$$

Finally, following the above formalization, for each input field f *Bijoux* produces a final set of disjoint, non-empty value ranges ($R^{f,final}$) and for each range it generates an instance of f inside that interval.

See for example, in Figure 6 and Figure 7, the generated data sets for covering the ETL process flow of our running example. We should mention at this point, that non conflicting constraints for the same field that is present in different paths and/or path combinations, can be merged and determine a single range (i.e., the intersection of all the ranges resulting

FINWIRE

PTS	RecType	Status	CoNameOrCIK
19880121171542	"SEC"	"ACTV"	"5609324496"
20160215210536	"SEC"	"CMPT"	"UPC CORP"

StatusType

ST_ID	ST_NAME
"ACTV"	"Active"
"CMPT"	"Complete"

DimCompany

CompanyID	Name	EffectiveDate	EndDate
"5609324496"	"TUD INC"	19681102185012	99991231000000
"1392258420"	"UPC CORP"	20011025102033	99991231000000

Figure 7: Generated datasets corresponding to the generated data

from the different paths). This way, under some conditions, the same value within that interval can be used for the coverage of different paths. As an example, in Figure 6, the fields *Status* and *ST_ID* that exist in both path combinations, all hold a constraint (ST_ID==Status). These can be merged into one single constraint, allowing for the generation of only one row for the table *StatusType* that can be used for the coverage of both path combinations, as long as both generated values for the field *Status* equal the generated value for the field *ST_ID* (e.g., "ACTV").

Following this idea, it can easily be shown that under specific conditions, the resulting constraints for the different path combinations from the application of our algorithm, can be further reduced, until they can produce a minimal set of datasets for the coverage of the ETL flow.

Data generation patterns must be further combined with other user-defined data generation parameters (e.g., selectivities, value distribution, etc.). We provide more details regarding this within our test case in Section 4.

3.6. Theoretical validation

We further provide a theoretical validation of our data generation process in terms of: the *correctness* of generated data sets (i.e., *path and flow coverage*).

A theoretical proof of the correctness of the *Bijoux* data generation process is divided into the three following components.

1. *Completeness of path traces.* Following from Algorithm 1, for each ETL graph node (i.e., datastores and operations, see Section 2.2) *Bijoux* builds path traces of all the paths reaching that node (e.g., see

Figure 4b). Formally, given that an ETL graph node can represent either an operation ($\mathbf{O}$), a source ($\mathbf{DS}_S$), or a target data store ($\mathbf{DS}_T$), we recursively formalize the existence of path traces as follows:

$$\forall v_i \in \mathbf{O} \cup \mathbf{DS}_T, \mathbb{PT}_{v_i} = \bigcup_{j=1}^{|\{v_j|v_j \prec v_i\}|} \Big\{ PT_{v_j}^{1} \cdot v_i, .., PT_{v_j}^{|\mathbb{PT}_j|} \cdot v_i \Big\}. \tag{4}$$

$$\forall v_i \in \mathbf{DS}_S, \mathbb{PT}_{v_i} = \{PT_{v_i}\}, PT_{v_i} = v_i. \tag{5}$$

Considering that ETL graph nodes are visited in a topological order (see Step 2 in Algorithm 1), the path traces of each ETL graph node are built after visiting all its predeceasing sub-paths. This guarantees that path traces of each node v_i are complete with regard to all its predecessors (i.e., $\{v_j|v_j \prec v_i\}$), hence the final path traces $\mathbb{FP}$ (i.e., path traces of target data store nodes) are also complete.

2. *Path coverage.* Having the complete path traces recorded in Algorithm 1, Algorithm 2 traverses a selected path (i.e., PT), with all its alternative *incidence paths*, and builds a data generation $Patern$ including a list of constraints over the input fields. Following from 1, this list of constraints is complete. Moreover, as explained in Section 3.5, *Bijoux* iteratively applies given constraints, and for each input field f produces a set of value ranges ($R^{f,final}$), within which the field values should be generated.

   Given the statements 1 - 3 in Section 3.5, *Bijoux* guarantees that the data generation stage applies all the constraints over the input fields when generating $R^{f,final}$, thus guaranteeing that the complete selected path will be covered.

   On the other side, if at any step of the data generation stage a result of applying a new constraint P_i leads to an empty set of value ranges, the collected list of constraints must be contradictory. Formally (following from statement 3 in Section 3.5):
   $(\exists R_i^{f,init}, R_{i-1}^{f} | R_i^f = \emptyset) \rightarrow \perp$.
   This further implies that the input ETL graph has contradictory path constraints that would lead to an unreachable sub-path, which could never be executed. As an additional functionality, *Bijoux* detects such behavior and accordingly warns the user that the input ETL flow is not correct.

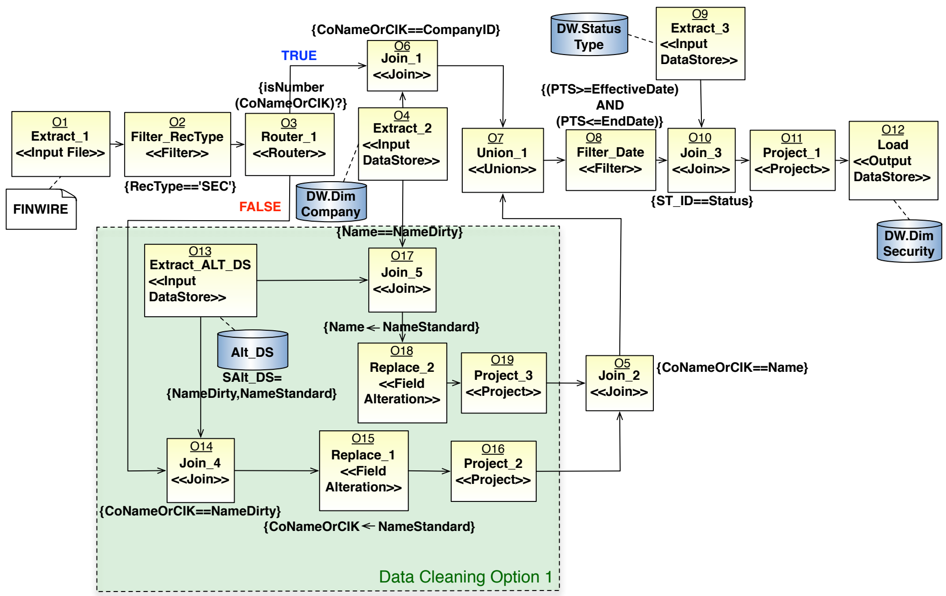


Figure 8: ETL flow for data cleaning, using a dictionary

3. *Flow coverage.* Following from 2, Algorithm 2 generates data that guarantee the coverage of a single path from $\mathbb{FP}$. In addition, if Algorithm 2 is executed for each final path $PT_i \in \mathbb{FP}$, it is straightforward that *Bijoux* will produce data that guarantee the coverage of the complete ETL flow (i.e., ETL graph), unless a constraints contradiction for an individual path has been detected.

4. Test case

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 regarding the evaluation of the quality of the flow. In this respect, we present in this section representative examples of how our framework can generate data, not only to enact specific parts of the ETL flow, but also to evaluate the performance and the data quality of these flow parts.

Going back to our running example (Figure 1), from now on referred to as *Flow_A*, we can identify a part of the flow that can be the source of data quality issues. That is, rows whose values for the field *CoNameOrCIK* are not numbers are matched with data about companies from the

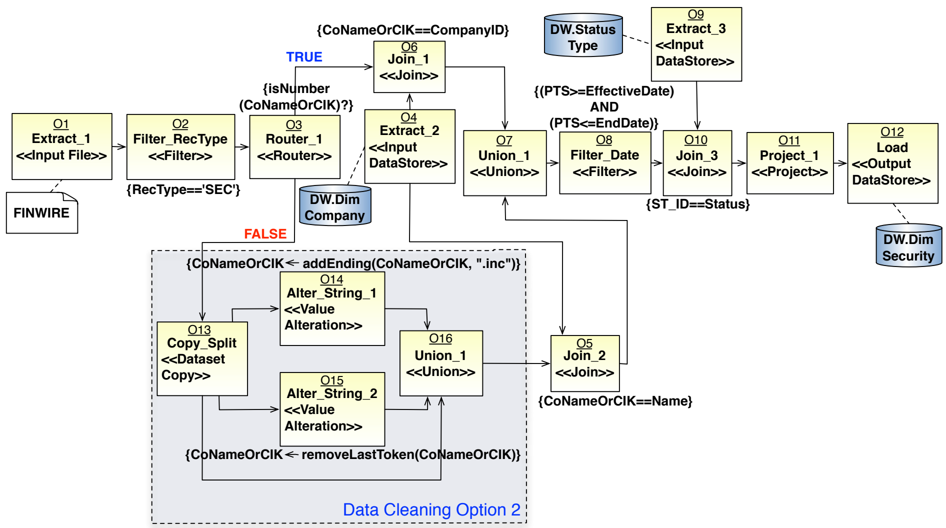


Figure 9: ETL flow for data cleaning, trying different string variations for the join key

DimCompany table, through an equi-join on the company name (CoNameOrCIK==Name). However, company names are typical cases of attributes that can take multiple values in different systems or even within the same system. For example, for a company *Abcd Efgh*, its name might be stored as “Abcd Efgh”, or followed by a word indicating its type of business entity (e.g., “Abcd Efgh Incorporated”) or its abbreviation with or without a comma (e.g., “Abcd Efgh Inc.” or “Abcd Efgh, Inc.”). It is also possible that it might be stored using its acronym (e.g., “ABEF”) or with a different reordering of the words in its name, especially when the two first words are name and surname of a person (e.g., “Efgh Abcd”). Moreover, there can be different uppercase and lowercase variations of the same string, combinations of the above-mentioned variations or even misspelled values.

Hence, there are many cases that the equi-join (CoNameOrCIK==Name) will fail to match the incoming data from the *FINWIRE* source with the rows from the *DimCompany* table, because they might simply be using a different variation of the company name value. This will have an impact on *data completeness*, since it will result in fewer rows being output to the DimSecurity than there should be.

To this end, we introduce here two more complex ETL flows (Figure 8 and Figure 9), which perform the same task as the running example, but include additional operations in order to improve the data quality of the output data. The ETL flow in Figure 8, from now on referred to as *Flow_B*, uses a dictionary (*Alt_DS*) as an alternative data source. This dictionary is assumed to have a very simple schema of two fields — *NameDirty* and *NameStandard*, to maintain a correspondence between different dirty variations of a company name and its standard name. For simplicity, we assume that for each company name, there is also one row in the dictionary containing the standard name, both as value for the *NameDirty* and the *NameStandard* fields. Operations *O14* and *O17* are used to match both the company names from the *FINWIRE* and the table, to the corresponding dictionary entries and subsequently, rows are matched with the standard name value being the join key, since the values for the join keys are replaced by the standard name values (*(Name←NameStandard)* and *(CoNameOrCIK←NameStandard)*).

Another alternative option for data cleaning is to try different variations of the company name value, by adding to the flow various string operations that alter the value of *CoNameOrCIK*. The ETL flow in Figure 9, from now on referred to as *Flow_C*, generates different variations of the value for *CoNameOrCIK* with operations *O14* and *O15*, who concatenate the abbreviation "inc." at the end of the word and remove the last token of the string, respectively. After the rows from these operations are merged through a Union operation (*O16*), together with the original *CoNameOrCIK* value, all these different variations are tried out to match with rows coming from *DimCompany*.

4.1. Evaluating the performance overhead of alternative ETL flows

In the first set of experiments, we implemented the three different ETL flows (*Flow_A*, *Flow_B* and *Flow_C*) using Pentaho Data Integration[4] and we measured their time performance by executing them on Kettle Engine, running on Mac OS X, 1.7 GHz Intel Core i5, 4GB DDR3 and keeping average values from 10 executions.

For each flow, we used *Bijoux* to generate data to cover only the part of the flow that was of interest, i.e., to cover the paths from Operations *O1* to *O12* who are covered by the rows that are evaluated as False by operation *O3*. Hence, one important advantage of our tool is that it can generate data to evaluate specific part of the flow, as opposed to random data generators (e.g., the TPC-DI data generator provided on the official website) who can only generate data agnostically of which part of the flow is being covered. This gives *Bijoux* not only a quality advantage, being able to evaluate the

[4] `http://www.pentaho.com/product/data-integration`

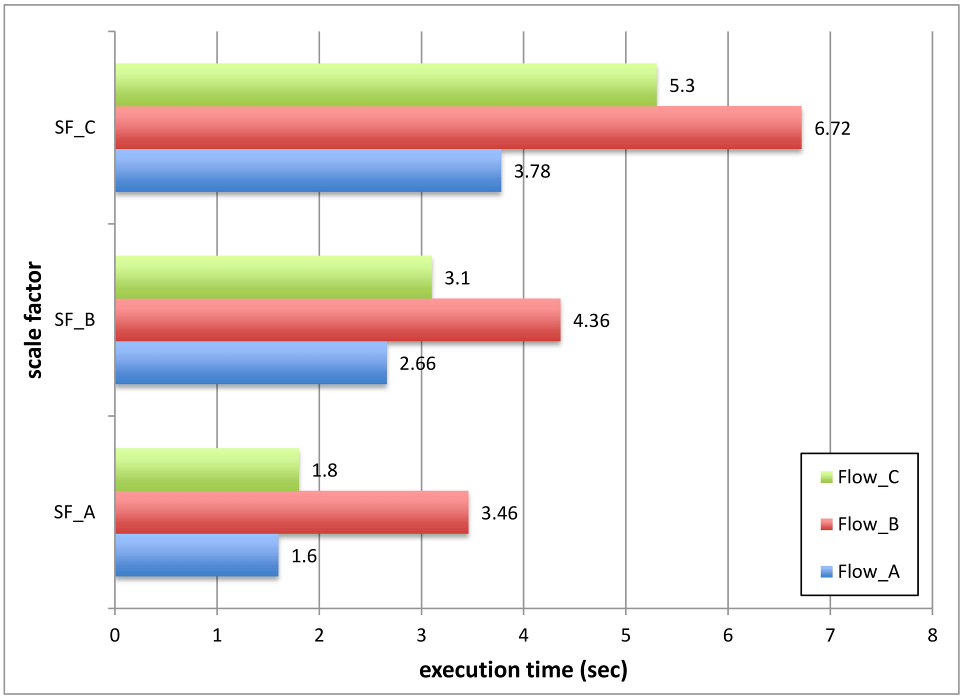


Figure 10: Performance evaluation of the flows using different scale factors

flow in greater granularity, but also a practical advantage, since the size of data that need to be generated can be significantly smaller. For instance, the TPC-DI data generator generates data for the FINWIRE file, only around $^1/_3$ of which are evaluated as *true* by the filter *RecType=='SEC'* and from them only around $^1/_3$ contains a company name instead of a number.

In order to generate realistic values for the company name fields, we used a catalog of company names that we found online [5] and we used *Bijoux* to generate data not only for the attributes that have been mentioned above, but for all of the attributes of the schemata of the involved data sources as defined in the TPC-DI documentation, so as to measure more accurate time results.

For each flow, we generated data of different size in order to evaluate how their performance can scale with respect to input data size, as shown in the below table, where we can see the number of rows for each data source for the three different scale factors (*SF*).

[5] `https://www.sec.gov/rules/other/4-460list.htm`

Data source →	FINWIRE	DimCompany	Alt_DS (for *Flow_B*)
SF_A	4000	4000	60000
SF_B	8000	8000	60000
SF_C	16000	16000	60000

For these experiments, for each flow we assumed selectivities that would guarantee the matching of all the rows in *FINWIRE* with rows in *DimCompany* and the results can be seen in Figure 10 For *Flow_C*.

As we expected, the results show an overhead in performance imposed by the data cleaning operations. It was also intuitive to expect that the lookup in the dictionary (*Flow_B*) would impose greater overhead than the string alterations (*Flow_C*). Nevertheless, some interesting finding that was not obvious is that as input data scale in size, the overhed of *Flow_B* appears to come closer and closer to the overhed of (*Flow_C*), which appears to become greater as input data size grows. We should notice at this point that our results regard the performance and scalability of a specific part of the flow – not the complete flow in general – which is a unique advantage of our approach, especially in cases of dealing with bottlenecks.

Consequently, we conducted experiments assuming different levels of input data dirtiness, by setting the selectivity of the different join operations for the different flows. The scenario we intended to simulate was a predefined percentage of different types of data dirtiness. In this respect, we considered four different types of dirtiness:

1. Missing the abbreviation "inc." at the end of the company name (Type_I)
2. A word (e.g., company type abbreviation) exists at the end of the name when it should not (Type_II)
3. The ending of the company name is mistakenly in an extended format (e.g., "incorporated' ' instead of "inc.") (Type_III)
4. Miscellaneous that cannot be predicted (e.g., "corp." instead of "inc." or misspelled names) (Type_IV)

We assumed that *Flow_A* cannot handle any of these cases (i.e., dirty names as an input for the *FINWIRE* source will fail to be matched to data coming from *DimCompany*); that *Flow_B* can solve all the cases for Type_I

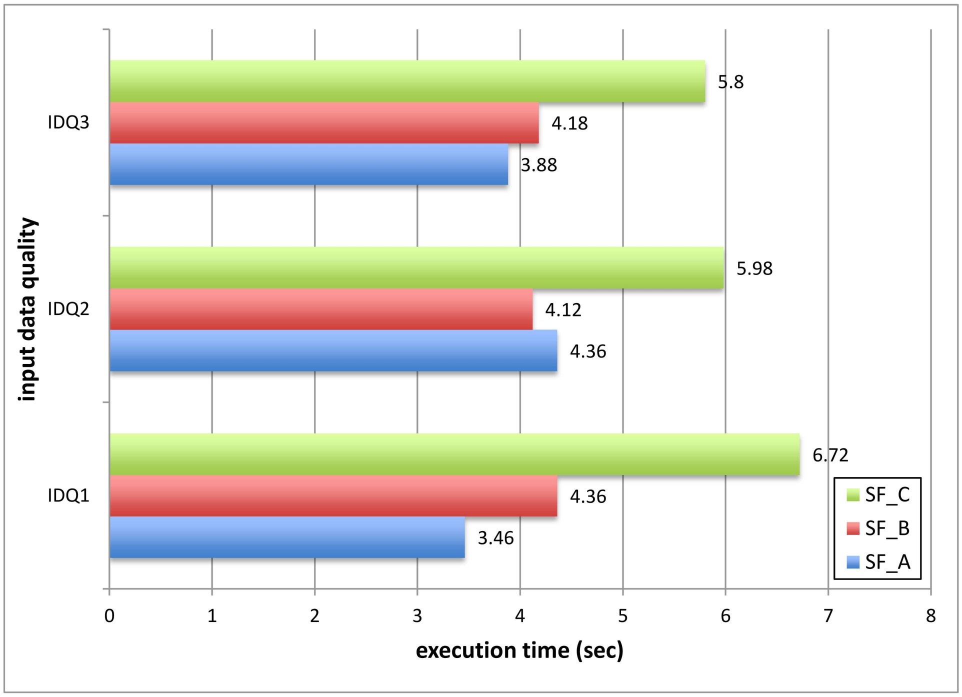


Figure 11: Performance evaluation of *Flow_B* using different levels of input data quality

and Type_III (i.e., there will be entries in the dictionary covering both of these types of dirtiness); and *Flow_C* can cover all the cases for Type_I and Type_II, because of the operation that it performs.

Thus, we generated data that were using real company names from the online catalog; we considered those names as the standard company names versions to generate data for the *DimCompany* source; and we indirectly introduced specified percentages of the different types of dirtiness, by setting a) the selectivities of the join operators and b) by manually generating entries in our dictionary (*Alt_DS*) that included all the names from the catalog together with their corresponding names manually transformed to Type_I and Type_II. The percentages of input data quality (IDQ) that were used for our experiments can be seen in the following table.

In Figure 11, we show how the performance of *Flow_B* scales with respect to different scale factors and data quality of input data. What is interesting about those results, is that the flow appears to be performing better when the levels of dirtiness of the input data are higher. This might appear counter-intuitive, but a possible explanation could be that less data

Dirtiness Type →	Type_I	Type_II	Type_III	Type_IV
IDQ1	0%	0%	0%	0%
IDQ2	1%	1%	3%	1%
IDQ3	2%	2%	6%	2%

(i.e., fewer rows) actually reach the extraction operation, keeping in mind that read/write operations are very costly for ETL flows.

4.2. Evaluating the data quality of alternative ETL flows

In the above-mentioned experiments, we evaluated the time performance of different flows, assuming that both data quality levels and data dirtiness characterization were a given. However, in order to evaluate an ETL flow with respect to the quality of the data cleaning that it can provide, it is not sufficient to only evaluate the time performance of different data cleaning options. To this end, in the second set of experiments, our goal was to evaluate which data cleaning option would produce the lowest levels of data incompleteness in the output data of the flow (*DimSecurity* table), using realistic datasets. In this respect, we used the company names from our catalog and for each of them we prepared a query to scrap the Freebase online database[6] and retrieve data about the company name and the known aliases of those names. Consequently, starting from 940 unique company names of our catalog, we were able to construct a dictionary that contained 2520 entries, each containing an alias of a company name and its corresponding standard name. We then used this dictionary as our Alt_DS dictionary; the standard names to populate the *DimCompany* table; and the names as they were on the catalog to populate the FINWIRE file.

Using *Bijoux* , we generated data that used *Flow_A* semantics in order to pass through the part of the flow that was of our interest and the dictionaries as mentioned above to generate realistic data. Despite the fact that it might appear as if the use of dictionaries devalues the use of our algorithm, in fact this is one strength of our approach — that it can be configured to generate data with different degrees of freedom, based on the constraints defined both by the flow semantics and the user. Therefore, it is possible to conduct such analysis, using a hybrid approach and evaluating the flows based on realistic

[6] `https://www.freebase.com/`

data. The contribution of our algorithm in this case is to generate, on one hand all the data for the different fields of the schemata that are required for the flow execution and to make sure, on the other hand that the generated rows will cover specific parts of the flow.

After executing Flow_B and Flow_C with these input data, we used the following measure for data completeness:

$DI = \%\_of\_missing\_entities\_from\_their\_appropriate\_storage$ [16]

The results for the two flows were the following:

$DI_{Flow\_B} = \frac{56}{940} * 100 \approx 6\%$

$DI_{Flow\_B} = \frac{726}{940} * 100 \approx 77\%$

According to these results, we can see a clear advantage of Flow_B regarding the data quality that it provides, suggesting that the performance overhead that it introduces, combined with potential cost of obtaining and maintaining a dictionary, might be worth undertaking, if data completeness is a goal of high priority.

We have explained above how the parametrization of our input data generation enables the evaluation of an ETL process and various design alterations over it, with respect to data quality and performance. Essentially, alternative implementations for the same ETL can be simulated using different variations of the data generation properties and the measured quality characteristics will indicate the best models, as well as how they can scale with respect not only to data size but also to data quality of the input data. Similarly, other quality characteristics can be considered, like *reliability* and *recoverability*, by adjusting the percentage of input data that result to exceptions and the selectivity of exception handling operations. In addition, we have shown through our examples how data properties in the input sources can guide the selection between alternative ETL flows during design time.

5. *Bijoux* performance evaluation

In this section, we report the experimental findings, after 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 (see Section 2.2). Users can also provide various parameters (see Figure 5) that can lead the process of data generation, which can refer to specific fields (e.g., *field distribution*), operations (e.g., *operation selectivity*) or general data generation parameters (e.g., *scale factors*).

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 provided by the users for the process description (i.e., distribution, operation selectivity, load size).

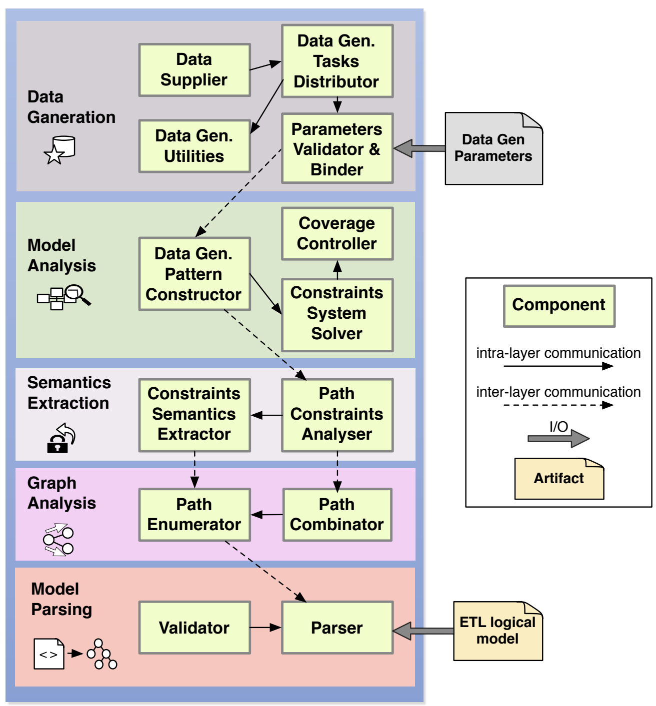


Figure 12: Bijoux prototype architecture

***Bijoux*'s architecture.** The *Bijoux*'s prototype is modular and based on a layered architecture, as shown in Figure 12. The four main layers implement the core functionality of the *Bijoux* algorithm (i.e., *graph analysis*, *semantics extraction*, *model analysis*, and *data generation*), while the additional bottom layer is responsible for importing ETL flows from corresponding files and can be externally provided and plugged to our framework (e.g., flow import plugin [13]). We further discuss all the layers in more detail.

- The bottom layer (*Model Parsing*) of the framework is responsible for parsing the model of the ETL process (*Parser* component) from the given logical representation of the flow (e.g., XML), and importing a DAG representation for the process inside the framework. In general, the *Model Parsing* layer can be extended with external parser plugins for handling different logical representations of an ETL process (e.g., [12, 13]). This layer also includes a *Validator* component to ensure syntactic, schematic and logical (e.g., cycle detection) correctness of the imported models.

- The *Graph Analysis* layer analyses the DAG representation of the ETL flow model. Thus, it is responsible for identifying and modeling all the ETL flow paths (*Path Enumerator* component; see Algorithm 1), as well as constructing all their possible combinations (*Path Combinator* component).

- The *Semantics Extraction* layer extracts relevant information needed to process the ETL flow. The information extracted in this layer (from the *Constraints Semantics Extractor* component) includes information about input datasets, operation semantics, order of operations, schema changes, and other parameters for data generation. This layer is also responsible for modeling constraints grouped by path (Path Constraints Analyzer; see Algorithm 2) to provide the required constructs for feasibility analysis and the construction of a data generation pattern to the layer above (Model Analysis).

- *Model Analysis* layer realizes the construction of a data generation pattern (*Data Gen. Pattern Constructor* component) that computes for each field (i.e., attribute), in each table, the ranges of values according to the extracted semantics of operations and their positioning within paths and path combinations. To this end, this layer includes the *Coverage Controller* component for implementing such analysis according to the set coverage goal (i.e., path coverage, flow coverage). In addition, it includes the *Constraints System Solver* component, which solves the systems of gathered constraints (e.g., system of logical predicates and equations over specified attributes) and returns the computed restrictions over the ranges.

- *Data Generation* layer controls the data generation stage according to the constraints (i.e., data generation patterns) extracted and analyzed in the previous layer, as well as the Data Gen. Parameters provided externally (e.g., distribution, selectivity). The *Parameters Validator & Binder* component binds the externally provided parameters to the ETL model and ensures their compliance with the data generation patterns, if it is possible. The *Data Gen. Tasks Distributor* component is responsible for managing the generation of data in a distributed fashion, where different threads can handle the data generation for different (pairs of) attributes, taking as input the computed ranges and properties (e.g., *generate 1000 values of normally distributed integers where 80% of them are lower than "10"*). For that purpose, it utilizes the *Data Gen. Utilities* component, that exploits dictionaries and random number generation methods. Finally, the *Data Supplier* component outputs generated data in the form of files (e.g., CSV files).

5.1. 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*) wrt. the growing complexity of the ETL model. The reason that we do not additionally test those quality aspects wrt. input load sizes is that such analysis is irrelevant according to the *Bijoux* algorithm. The output of the analysis phase is a set of ranges and data generation parameters for each attribute. Hence, the actual data generation phase does not depend on the efficiency of the proposed algorithm, but instead can be realized in an obvious and distributed fashion. Thus, we present our results from experiments that span across the phases of the algorithm up until the generation of ranges for each attribute. We performed the performance testing considering several ETL test cases, which we describe in what follows.

Our experiments were carried under an OS X 64-bit machine, Processor Intel Core i5, 1.7 GHz and 4GB of DDR3 RAM. The test cases consider a subset of ETL operations, i.e., *Input DataStore, Join, Filter, Router, UDF, Aggregation* and *Output DataStore*. Based on the TPC-H benchmark[7], our basic scenario is an ETL process, which extracts data from a source relational database (TPC-H DB) and after processing, loads data to a data warehouse (DW) and can be described by the following query: *Load in the DW all the suppliers in Europe together with their information (phones, addresses etc.), sorted on their revenue and separated by their account balance*

[7] http://www.tpc.org/tpch/

(*either low or high)*, as can be seen in Fig. 13.

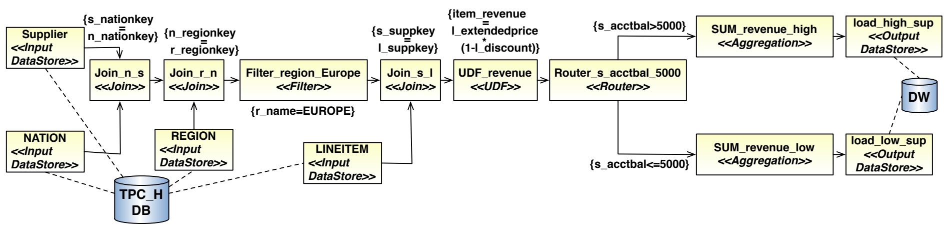


Figure 13: Basic scenario ETL process for experiments

The tables that are used from the source database are *Supplier*, *Nation*, *Region* and *Lineitem*. After *Supplier* entries have been filtered to keep only suppliers in Europe, the revenue for each supplier is calculated based on the supplied lineitems and subsequently, they are sorted on revenue, separated by their account balance and loaded to different tables in the DW. Starting from the basic scenario, we use *POIESIS* [17], a tool for ETL Process redesign that allows for the automatic addition of flow patterns on an ETL model. Thus, we create other, more complex, synthetic ETL flows. The motivation for using tools for automatic ETL flow generation stems from the fact that obtaining real world ETL flows covering different scenarios with different complexity and load sizes is hard and often impossible.

Scenarios creation. Starting from this basic scenario, we create more complex ETL flows by adding additional operations, i.e., *Join, Filter, Input DataStore, Project* in various (random) positions on the original flow. We add two different Flow Component Patterns (FCP) [17] on the initial ETL flow in different cardinalities and combinations. The first pattern — *Join* — adds 3 operations every time it is applied on a flow: one Input DataStore, one Join and one Project operation in order to guarantee matching schemata; the second pattern — *Filter* — adds one Filter operation with a random (inequality) condition on a random numerical field (i.e., attribute).

We iteratively create 5 cases of different ETL flow complexities and observe the *Bijoux*'s execution time for these cases, starting from the basic ETL flow:

- *Case 1.* Basic ETL scenario, consisting of twenty-two (22) operations, as described above (before each join operation there exists also one joining key sorting operation which is not shown in Fig. 13, so that the flow is executable by most popular ETL engines).
- *Case 2.* ETL scenario consisting of 27 operations, starting from the

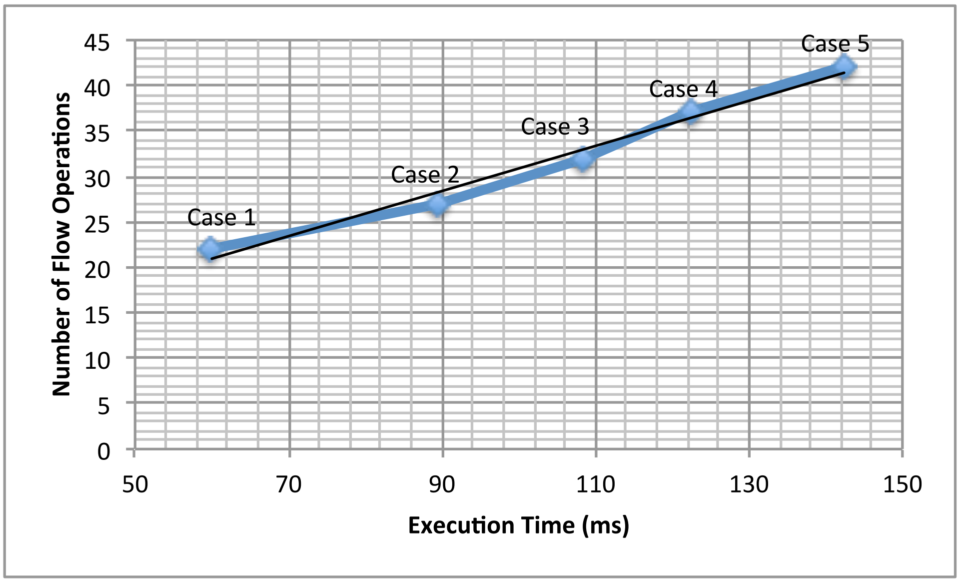


Figure 14: Linear trend of constraints extraction time wrt. the increasing number of operations (ETL flow complexity)

basic one and adding an additional *Join* FCP and 2 *Filter* FCP to the flow.

- *Case 3.* ETL scenario consisting of 32 operations, starting from the basic one and adding 2 additional *Join* FCP and 4 *Filter* FCP to the flow.
- *Case 4.* ETL scenario consisting of 37 operations, starting from the basic one and adding 3 additional *Join* FCP and 6 *Filter* FCP to the flow.
- *Case 5.* ETL scenario consisting of 42 operations, starting from the basic one and adding 4 additional *Join* FCP and 8 *Filter* FCP to the flow.

5.2. Experimental results

We measure the average execution time of the path enumeration, extraction and analysis phase for the above 5 scenarios covering different ETL flow complexities.

Figure 14 illustrates the increase of execution time when moving from the simplest ETL scenario to a more complex one. As can be observed, execution time appears to follow a linear trend wrt. the number of operations of the ETL flow (i.e., flow complexity). This can be justified by the efficiency of our graph analysis algorithms and by the extensive use of indexing techniques (e.g., hash tables) to store computed properties for each operation and field, perhaps with a small overhead on memory usage. This result might appear contradictory, regarding the combinatorial part of our algorithm, computing and dealing with all possible path combinations. Despite the fact that it imposes factorial complexity, it is apparent that it does not constitute a performance issue for ETL flows of such complexity. To this end, the solution space is significantly reduced by i) our proposed greedy evaluation of the feasibility of a pattern every time it is updated and ii) by disregarding path combinations that do not comply to specific rules, e.g., *when considering path coverage, every input of a joining operation involved in any path of a path combination must be flowed (crossed by) at least one other path of that combination.*

6. Related Work

Software development and testing. In the software engineering field, *test-driven development* has studied the problem of software development by creating tests cases in advance for each newly added feature in the current software configuration [18]. However, in our work, we do not focus on the design (i.e., development) of ETL processes per se, but on automating the evaluation of quality features of the existing designs. We analyze how the semantics of ETL processes entail the constraints over the input data, and then consequently create the testing data. Similarly, the problem of constraint-guided generation of synthetic data has been also previously studied in the field of software testing [5]. 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.).

Data generation for relational databases. Moving toward the database world, [19] 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, the approach generates a set of constraints, which can be given to existing constraints solvers. If the constraints are satisfiable, a desired database instances are obtained. Similarly, for testing the correctness of relational DB systems, a study in [20] proposes a semi-automatic approach for populating the database with meaningful data that satisfy database constraints. Work in [21] 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 [21], the authors generate synthetic database instance from scratch, rather than by modifying the existing one. Furthermore, [22] proposes 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 [23] 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 [24] 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 systems. 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 (e.g., complex user defined functions, pivot/unpivot). 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.

Benchmarking data integration processes. In a more general context, both research and industry are particularly interested in benchmarking ETL and data integration processes in order to evaluate process designs and compare different integration tools (e.g., [25, 26]). Both these works note the lack of a widely accepted standard for evaluating data 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 scenarios to evaluate the process design 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.

General data generators. Other approaches have been working on providing data generators that are able to simulate real-world data sets for the purpose of benchmarking and evaluation. [27] 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 [28], the authors present a tool called Big Data Generator Suite (BDGS) for generating Big Data meanwhile preserving the 4V characteristics of Big Data [8]. 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 [30], 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., [31]) 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 multidimensional 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, the data generation is not led by the constraints that the operations entail over the input data, hence they cannot be customized for evaluating different quality features of ETL-like processes.

Process simulation. 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 [32], we

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

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 [33]. [34] 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 [35] 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 [36, 3].

7. 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 different quality factors of ETL process models (e.g., data completeness, reliability, 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 framework. The 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*, especially considering larger volumes of generated data.

As an immediate future step, we plan on additionally validating and exploiting the functionality of this approach in the context of quality-driven ETL process design and tuning, as explained in our test case scenario.

8. Acknowledgements

This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate “Information Technologies for Business Intelligence - Doctoral College” (IT4BI-DC).

References

1. Barbacci, M., Klein, M.H., Longstaff, T.A., Weinstock, C.B.. Quality attributes. Tech. Rep.; CMU SEI; 1995.

2. Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.. Qox-driven etl design: reducing the cost of etl consulting engagements. In: *SIGMOD Conference*. 2009, p. 953–960.

3. Theodorou, V., Abelló, A., Lehner, W., Thiele, M.. Quality measures for ETL processes: from goals to implementation. *Concurrency and Computation Practice and Experience: Version of record online* 2016;.

4. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.. *Fundamentals of Business Process Management*. Springer; 2013. ISBN 978-3-642-33142-8.

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

6. Strange, K.H.. ETL Was the Key to This Data Warehouse's Success. March 2002. Gartner Research, CS-15-3143.

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

8. Vassiliadis, P., Simitsis, A., Baikousi, E.. A taxonomy of etl activities. In: *DOLAP*. 2009, p. 25–32.

9. Vassiliadis, P., Simitsis, A., Skiadopoulos, S.. Conceptual modeling for ETL processes. In: *DOLAP*. 2002, p. 14–21.

10. Muñoz, L., Mazón, J.N., Pardillo, J., Trujillo, J.. Modelling etl processes of data warehouses with uml activity diagrams. In: *OTM Workshops*. 2008, p. 44–53.

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

12. Wilkinson, K., Simitsis, A., Castellanos, M., Dayal, U.. Leveraging business process models for etl design. In: *ER*. 2010, p. 15–30.

13. Jovanovic, P., Simitsis, A., Wilkinson, K.. Engine independence for logical analytic flows. In: *ICDE*. 2014, p. 1060–1071.

14. Hueske, F., Peters, M., Sax, M., Rheinländer, A., Bergmann, R., Krettek, A., et al. Opening the black boxes in data flow optimization. *PVLDB* 2012; **5**(11):1256–1267.

15. Tziovara, V., Vassiliadis, P., Simitsis, A.. Deciding the physical implementation of etl workflows. In: *Proceedings of the ACM Tenth International Workshop on Data Warehousing and OLAP*; DOLAP '07. 2007, p. 49–56.

16. Simitsis, A., Vassiliadis, P., Dayal, U., Karagiannis, A., Tziovara, V.. Benchmarking etl workflows. In: *Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*. 2009, p. 199–220.

17. Theodorou, V., Abelló, A., Thiele, M., Lehner, W.. POIESIS: a tool for quality-aware ETL process redesign. In: *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.* 2015, p. 545–548.

18. Beck, K.. *Test-driven development: by example*. Addison-Wesley Professional; 2003.

19. Zhang, J., Xu, C., Cheung, S.C.. Automatic generation of database instances for white-box testing. In: *COMPSAC*. 2001, p. 161–165.

20. Chays, D., Dan, S., Frankl, P.G., Vokolos, F.I., Weber, E.J.. A framework for testing database applications. In: *ISSTA*. 2000, p. 147–157.

21. Arasu, A., Kaushik, R., Li, J.. Data generation using declarative constraints. In: *SIGMOD Conference*. 2011, p. 685–696.

22. Binnig, C., Kossmann, D., Lo, E., Özsu, M.T.. Qagen: generating query-aware test databases. In: *SIGMOD Conference*. 2007, p. 341–352.

23. Houkjær, K., Torp, K., Wind, R.. Simple and realistic data generation. In: *VLDB*. 2006, p. 1243–1246.

24. Lakhotia, K., Harman, M., McMinn, P.. A multi-objective approach to search-based test data generation. In: *GECCO*. 2007, p. 1098–1105.

25. Simitsis, A., Vassiliadis, P., Dayal, U., Karagiannis, A., Tziovara, V.. Benchmarking etl workflows. In: *TPCTC*. 2009, p. 199–220.

26. Böhm, M., Habich, D., Lehner, W., Wloka, U.. Dipbench toolsuite: A framework for benchmarking integration systems. In: *ICDE*. 2008, p. 1596–1599.

27. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.. Quickly Generating Billion-Record Synthetic Databases. In: *SIGMOD Conference*. 1994, p. 243–252.

28. Ming, Z., Luo, C., Gao, W., et al. Bdgs: A scalable big data generator suite in big data benchmarking. *CoRR* 2014;**abs/1401.5465**.

29. Luo, C., Gao, W., Jia, Z., Han, R., Li, J., Lin, X., et al. Handbook of BigDataBench (Version 3.1) - A Big Data Benchmark Suite. 2014. Last accessed: 13/05/2015; URL `http://prof.ict.ac.cn/BigDataBench/wp-content/uploads/2014/12/BigDataBench-handbook-6-12-16.pdf`.

30. Rabl, T., Frank, M., Sergieh, H.M., Kosch, H.. A data generator for cloud-scale benchmarking. In: *TPCTC*. 2010, p. 41–56.

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

32. 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*. McGraw-Hill; 1998, p. 311–320.

33. Law, A.M., Kelton, W.D., Kelton, W.D.. *Simulation modeling and analysis*; vol. 2. McGraw-Hill; 1991.

34. Becker, J., Kugeler, M., Rosemann, M.. *Process Management: a guide for the design of business processes: with 83 figures and 34 tables*. Springer; 2003.

35. Jansen-Vullers, M., Netjes, M.. Business process simulation–a tool survey. In: *Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*; vol. 38. 2006, p. –.

36. Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.. QoX-driven ETL design: reducing the cost of ETL consulting engagements. In: *SIGMOD*. 2009, p. 953–960.