# Data Generation for the Simulation of Artifact-Centric Processes

**Emona Nakuçi**

Supervisor: Alberto Abelló
Advisors: Petar Jovanovic and Vasileios Theodorou

**MASTER ON INFORMATION TECHNOLOGIES FOR BUSINESS INTELLIGENCE**

Universitat Politècnica de Catalunya
Barcelona
July 2014

# ACKNOWLEDGEMENTS

I would like to dedicate few words to the people who helped me accomplish this thesis, which represents not only my work, but also that of those who supported me throughout this research.

Hence, it gives me great pleasure in acknowledging the support and help of my advisors Petar Jovanovic and Vasileios Theodorou for their continuous guidance and useful advices.

I would like to express my deepest gratitude to my supervisor, Alberto Abelló for believing in me and helping me think deeply about the problem.

Many thanks go to my dearest friends who always made me smile and think positively even in the hardest times.

Finally, my special thanks go to my wonderful family to whom I dedicate this thesis. I thank my parents for always being by my side, although often virtually. I would like to especially thank my brother, Enor, for always believing in me and encouraging me to achieve more. None of this would have been possible without your love and support.

# TABLE OF CONTENTS


**ACKNOWLEDGEMENTS** **iii**

**ABSTRACT** **1**

**1 INTRODUCTION** **2**

1.1 Problem Statement . . . . . 2

1.2 Motivation . . . . . 2

1.3 Objectives of the Study . . . . . 3

1.4 Scientific Contribution . . . . . 4

1.5 Outline of the Thesis . . . . . 6

**2 BACKGROUND** **8**

2.1 Business Processes . . . . . 10

2.1.1 BPM . . . . . 10

2.1.2 BPMN . . . . . 11

2.2 Data Processes . . . . . 11

2.2.1 ETL Processes . . . . . 11

2.2.2 ETL Modeling . . . . . 13

2.3 Quantitative and Qualitative Analysis of Business Processes . . . . . 18

**3 RELATED WORK** **22**

3.1 ETL Tools . . . . . 22

3.2 ETL Evaluation and Optimization . . . . . 23

3.3 ETL Process Benchmarking . . . . . 24
3.4 Data Generators for Benchmarking Purposes . . . . . 26
3.5 Constraint-based Data Generation . . . . . 31
3.6 BPMN Simulators . . . . . 33

**4 FORMALIZING THE PROBLEM OF DATA GENERATION 37**

4.1 Defining the ETL Operation . . . . . 38
4.2 Data Integration Tools Selection . . . . . 40
4.3 Literature Review on ETL Taxonomies . . . . . 40
4.4 Proposed ETL Taxonomy . . . . . 42
4.4.1 Defining List of ETL Operations . . . . . 42
4.4.2 Proposed ETL Taxonomy . . . . . 46
4.4.3 Value Level Operations . . . . . 53
4.4.4 Attribute Level Operations . . . . . 53
4.4.5 Tuple Level Operations . . . . . 54
4.4.6 Dataset Level Operations . . . . . 55
4.4.7 Schema Level Operations . . . . . 56
4.4.8 Relation Level Operations . . . . . 57
4.5 ETL Operation Semantics Definition . . . . . 58
4.6 Solution Limitations . . . . . 63
4.6.1 Atomic Operations . . . . . 63
4.6.2 Logical Predicates . . . . . 63
4.6.3 Custom-defined Transformations . . . . . 64
4.6.4 Tool-specific Operations . . . . . 64
4.7 Extensibility . . . . . 65

**5 ARCHITECTURE DESIGN 67**

5.1 ETL Data Generation Architecture . . . . . 67
5.1.1 Input . . . . . 67
5.1.2 Model Parsing . . . . . 69
5.1.3 Model Extraction . . . . . 70
5.1.4 Model Analysis . . . . . 71

5.1.5 Data Generation . . . . . 71
5.1.6 Presentation Layer . . . . . 72
5.1.7 Controller . . . . . 73
5.1.8 Output . . . . . 73
5.2 Data Generation Algorithm . . . . . 74
5.2.1 Process Flow . . . . . 74
5.2.2 Algorithm . . . . . 75
5.2.3 Algorithm Illustration . . . . . 80

**6 PROTOTYPING 92**
6.1 Implementation . . . . . 92
6.1.1 Technologies Used . . . . . 93
6.1.2 Implemented Architecture . . . . . 94
6.2 Experimental Setup and Testing . . . . . 99
6.3 Experimental Results . . . . . 101
6.4 Discussion . . . . . 104

**7 CONCLUSIONS AND FUTURE WORK 106**
7.1 Contribution . . . . . 106
7.2 Future Work . . . . . 107

**BIBLIOGRAPHY 109**

**APPENDIX 116**
A.1 ETL Operation Semantics Definition . . . . . 116

# LIST OF FIGURES


1.1 Thesis positioning and contribution . . . . . 5

2.1 An ETL process . . . . . 12
2.2 Logical modeling layer of ETL flow. Example taken from [1] . . . . . 14
2.3 ETL flow modeled in BPMN. Example taken from [2] . . . . . 16
2.4 Performance taxonomy taken from [3] . . . . . 19

3.1 BP simulation workflow . . . . . 34

4.1 Example of an ETL atom and molecule taken from [4] . . . . . 39
4.2 Adapting taxonomy from [4] to our scope . . . . . 47
4.3 Relation access level taxonomy . . . . . 48
4.4 Proposed ETL taxonomy . . . . . 52
4.5 Value-level operations . . . . . 53
4.6 Attribute-level operations . . . . . 54
4.7 Tuple-level operations . . . . . 55
4.8 Dataset-level operations . . . . . 56
4.9 Schema-level operations . . . . . 57
4.10 Relation-level operations . . . . . 58

5.1 ETL data generator framework architecture . . . . . 68
5.2 Model parsing layer . . . . . 69
5.3 Model extraction layer . . . . . 70

5.4 Model analyzer layer . . . . . 71
5.5 Data generation layer . . . . . 72
5.6 Presentation layer . . . . . 73
5.7 Data generation process flow . . . . . 74
5.8 ETL flow example . . . . . 80
5.9 Attribute parameters structure, AP . . . . . 81
5.10 Operation parameters structure, OP . . . . . 81
5.11 Table of constraints, TC . . . . . 82
5.12 First iteration . . . . . 84
5.13 First iteration result . . . . . 85
5.14 Second iteration . . . . . 85
5.15 Second iteration result . . . . . 86
5.16 Third iteration . . . . . 87
5.17 Third iteration result . . . . . 88
5.18 Fourth iteration . . . . . 88
5.19 Fourth iteration result . . . . . 89
5.20 Final result . . . . . 90

6.1 Referenced architecture . . . . . 94
6.2 Example of an expression tree . . . . . 95
6.3 Implementation architecture . . . . . 97
6.4 Basic ETL scenario . . . . . 100
6.5 Incremental ETL scenario . . . . . 101
6.6 Generation time wrt flow complexity . . . . . 102
6.7 Generation time wrt load size . . . . . 103
6.8 Linear trend of the data generation performance for flow with complexity 4 and 9 as load size increases . . . . . 103

# LIST OF TABLES

2.1 BI definitions . . . . . 9

3.1 Data generator tools' feature support . . . . . 29

4.1 Source and target operations . . . . . 41

4.2 List of operations considered in the framework . . . . . 42

4.3 Transformations provided by four main ETL tools . . . . . 45

4.4 Table of operation semantics notations . . . . . 60

1 Table of ETL operations semantics . . . . . 116

# LIST OF ABBREVIATIONS

| | |
|---|---|
| BI | Business Intelligence |
| BPEL | Business Process Execution Language |
| BPM | Business Process Management |
| BPMN | Business Process Modeling Notation |
| BPS | Business Process Simulation |
| CSV | Comma-Separated Values |
| DAG | Directed Acyclic Graph |
| DBMS | Database Management System |
| DW | Data Warehouse |
| ETL | Extract, Transform and Load |
| OWB | Oracle Warehouse Builder |
| SOA | Service-Oriented Architecture |
| SSIS | SQL Server Integration Services |
| SQL | Structured Query Language |
| UML | Unified Modeling Language |
| XML | Extensible Markup Language |

# ABSTRACT

Increasing need for application benchmarking and testing purposes requires large amounts of data. However, obtaining realistic data from the industry for testing purposes, is often impossible due to confidentiality issues and expensive data transfer over the network i.e., Internet. Hence, there is a gap between the need to benchmark and the lack of a common testing environment to achieve it.

The scope of this thesis is to contribute in narrowing the above presented gap, by introducing a theoretical framework of data generation for the simulation of data processes. Therefore, we aim at generating input data and hence, providing a common testing environment for testing and evaluating data processes. Specifically, we focus on generating data for ETL data processes by analyzing the semantics of the flow. The motivation comes from the fact that ETL processes are often time-consuming and error prone. Therefore, it is of high importance to evaluate and benchmark them, in order to identify bottlenecks and constantly improve their performance.

Moreover, we introduce a layered architecture design for developing a prototype of the ETL data generation framework. In addition, we present a pilot tool developed for implementing the ETL data generation framework following the proposed architecture and the ETL semantics principle. As a conclusion to our work, we introduce the data generation approach and moreover show its feasibility to generate workload scenarios useful for testing and benchmarking ETL processes.

# 1 INTRODUCTION

## 1.1 Problem Statement

Increasing need for application benchmarking and testing purposes requires large amount of data. Obtaining these test data is often impossible due to confidentiality issues and expensive transfer over the network. This thesis aims at providing an approach to generate synthetic data for ETL workflows. The incentive for this work comes from the difficulties to obtain full running cases of real-world business processes and from the high needs to evaluate, compare and benchmark ETL processes.

By analyzing the data flow transformation semantics, we provide the means to automatically generate representative input data for data processes, that can successfully replay the flow. During the experimental work we focus on ETL data processes. The main motivation of this work comes from the fact that ETL processes are often time-consuming and error prone. Thus, it is important to identify bottlenecks in these data processes and improve their performance.

## 1.2 Motivation

Data warehousing (DW) is a concept that dates back to the early 90s. Since then, attention has been devoted to the modeling of ETL process that expresses the flow of data from operational systems to the data warehouse. ETL is a centric process of DW since the quality and accuracy of information in a DW highly depends on the ETL flow design. Such process is very expensive in terms of resources used

and design time. To facilitate its design, there are many ETL tools that allow the modeling and execution via user-friendly interfaces. However, each tool is very specific and has developed its own modeling techniques to design the ETL flow. They differ in many characteristics such as user interface, underlying technologies, ETL modeling syntax and operation palette etc. Consequently, there is no agreed standard on the design and representation of ETL workflows.

The purpose of our data generation project is to generate data specific to ETL processes. Currently there exists a vast plethora of tools that aim at generating data, but none of them addresses particularly ETL processes. The majority is dedicated to generating data for DBMS and serve the purpose of generating synthetic or realistic data for testing and benchmarking reasons.

Our project is along the same line as current data generator tools already in the market, with the particularity though to address ETL processes only. The motivation comes from the lack of a standard ETL modeling representation and lack of widely accepted benchmarking principles. Up to now, still many differences exist when designing ETL flows, partially due to tools differences, designer expertize and company-specific objectives, which in turn, is still left on the hands of the designer rather than following a standard framework with predefined principles.

## 1.3 Objectives of the Study

In order to contribute to fill the gap between the need to evaluate, test and benchmark ETL processes and the lack of a common solid environment to compare them, we have worked on this master project with the aim to generate synthetic test data specifically for the ETL flow.

The scope of this thesis is to introduce a theoretical framework for generating test data for ETL processes, by resolving the semantics of the flow, and moreover, support its feasibility by developing a prototype to achieve this goal. This framework is aimed to be extendable and configurable for different flow characteristics (e.g.,selectivity, distribution etc.). Our objective is driven by the necessity for evaluating and benchmarking ETL processes.

To successfully achieve our goal, we have defined four main objectives:

- Define a complete list of ETL operations that are typical of most ETL flows and supported by current data integration tools available. To accomplish this objective, we demonstrate the examination we did on four major data integration tools and how we came up with the final list of ETL operations.
- Categorize the list of ETL operations defined previously in a comprehensive taxonomy based on the ETL semantics principles that will help and guide us into the data generation process.
- Represent the semantics of each operation in a formalized way, simplistic but yet expressive, by using common language (i.e., first order logic).
- Show the feasibility of our data generation framework by implementing a prototype tool that generates input data for basic ETL flows covering a selected sample of ETL operations from the defined list.

## 1.4 Scientific Contribution

Both academic research communities and industrial ones have shown interest with regards to efforts for benchmarking ETL processes in order to create a common environment for evaluating and improving them. On the one hand, researchers are interested in developing standardized methods for building and comparing prototypes. On the other hand, customers are interested in having a solid platform for comparing different ETL tools before acquiring one, while industrial vendors are interested in measuring the performance, reliability of their ETL products and knowing how and what part to improve. This is particularly necessary when considering the estimated time and cost devoted to the purchase and design of ETL processes in a given organization. Interesting figures presented in [5] show high amounts of time and money dedicated to the ETL initiative as a whole, which comprises 30% of effort and expenses in the budget of the DW, 55% of the total costs of DW runtime and 80% of the development time in a DW project.

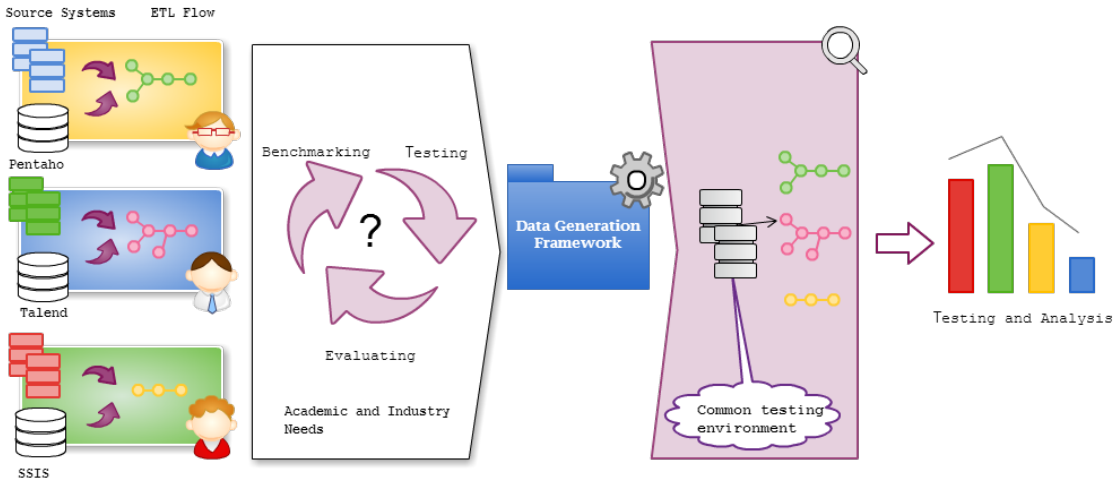



Figure 1.1: Thesis positioning and contribution

In Figure 1.1, we illustrate the existing gap between the necessity to compare, evaluate and benchmark ETL processes and the problems that halt its feasibility. On the one hand, we face the problem of lacking a formalization of ETL process representation, which depends on the ETL tool used to design it, and also on the designer's expertize, as illustrated in the left part of the picture. On the other hand, there is no common testing environment to actually perform the evaluation. So, what we propose is a data generation framework solution that will generate common workloads. Finally, by providing a common testing environment, we contribute in helping the testing, evaluation and benchmarking initiatives of ETL processes.

To summarize, our contribution lies in helping to narrow the gap between the necessity to compare, evaluate and benchmark ETL processes and the lack of a common environment to achieve it. We contribute in creating a framework to generate ETL data that behaves according to the semantics of the flow and thus, simulates its behaviour as if they were realistic data. Moreover, this framework is configurable for different model criteria and extendable based on user requirements. With this thesis project, we provide a common layer of workload that can be used during the simulation of ETL flows for evaluation and benchmarking purposes.

## 1.5 Outline of the Thesis

Chapter 1 - Introduction

In this chapter we introduce the scope of our research study. We state the problem we are raising and how we tackle it, by introducing the strategy and setting up the goals and objectives that we aim to achieve.

Chapter 2 - Background

In the second chapter we discuss the background information that is necessary to set up the knowledge to later present our work in details. We talk about business processes and specifically data processes with ETL as an example, and current approaches and research studies on their quantitative and qualitative analysis. We also introduce the BPM and BPMN concepts, the relation among them and how they are important to our topic.

Chapter 3 - Related Work

In this chapter we give a brief introduction to the specific ETL tools that are examined for the purpose of our study. In continuation, we discuss related work and literature, with respect to ETL evaluation and optimization and current benchmarking initiatives proposed so far. In addition, we also examine the literature regarding current data generator tools for benchmarking purposes to see if they tackle the data generation problem for ETL processes. Moreover, we inspect the related research study on BPMN simulation tools under the data generation perspective.

Chapter 4 - Formalizing the Problem of Data Generation

The aim of this chapter is to provide the theoretical framework of our ETL data generation solution. More concretely, we present the procedure we followed when inspecting ETL tools and their plethora of operations in order to define the final list of ETL operations considered in our research. In addition, we present the approach followed for classifying them towards building an ETL taxonomy. Moreover, we introduce a formalization of the ETL operation semantics using expressive notations from the first order logic. This chapter gives a high level view of the proposed data generation solution.

Chapter 5 - Architecture Design and Implementation

This chapter is devoted to the layered architecture design that we propose for our

data generation framework. We explain in details each layer and the corresponding components, the functionalities they support and how they communicate with each other. Important part of this chapter is presenting the algorithm we develop and propose for the data generation approach.

Chapter 6 - Prototyping

This chapter introduces the prototype we implemented following the theoretical framework and the layered architecture design proposed. First of all, we present the technological environment used to implement and test the prototype. And then, we discuss the technical details of the prototype implemented, its modules and functionality. Lastly, we show the results of the experiments carried during the testing phase.

Chapter 7 - Conclusions and Future Work

The last chapter of the thesis presents the final conclusions and gives insights on the benefits and contribution of our work into other future studies.

# 2 BACKGROUND

Since the first definition of Business Intelligence or shortly referred to as BI, in the IBM journal of 1958 by Hans Peter Luhn [6], it has evolved and turned into a very "hot topic" in the recent years. The concept of BI given by Luhn back then was:

"*The ability to apprehend the interrelationships of presented facts in such a way as to guide action towards a desired goal.*"

Meanwhile, today this concept has been extended more by referring also to the tools and technologies utilized in order to perform the process analysis. Gartner Group[1] defines BI as:

"*...an umbrella term that includes the applications, infrastructure and tools, and best practices that enable access to and analysis of information to improve and optimize decisions and performance.*"

An interesting study on the evolution of BI as a concept has been carried in this Doctorate Thesis [7]. As the author points out, the concept of BI has evolved from one-dimensional considering it as a process of analyzing companies' raw data, into a more complete multi-dimensional concept including also the technology used during the process as well as the final outcome which is the knowledge gained from the process.

Below we present graphically the three perspectives of BI concepts starting from the mono-dimensional one which considers it mainly as a process only, and evolving later to a multi-dimensional concept that includes in the BI definition also the set

[1] http://www.gartner.com/it-glossary/business-intelligence-bi/

of technologies supporting it as well as the final outcome.

| BI Perspective | BI Definition |
|---|---|
| **BI as a process** | "Business Intelligence (BI) can be defined as the process of turning data into information and then into knowledge." [8] |
| **BI as a set of technologies** | "Business intelligence (BI) is a broad category of technologies, applications, and processes for gathering, storing, accessing, and analyzing data to help its users make better decisions." [9] "Business intelligence encompasses a set of tools, techniques, and processes to help harness this wide array of data and allow decision makers to convert it to useful information and knowledge." [10]. |
| **BI as a product** | This definition focuses on the final outcome which is the information and knowledge obtained after the process analysis, which is important for the strategic decision making. The obtained BI outcome (knowledge) is generally presented in the form of a report, spreadsheet, table, graph, metric or an integrated version of the above called scorecards and dashboards. |

Table 2.1: BI definitions

Nevertheless, despite the differences between the above BI perspectives, all these standpoints share same BI principles:

1. BI centric activity is the process of gathering, analyzing, processing data and information and producing knowledge

2. BI consists of a set of underlying technologies that help in the above mentioned processes

3. BI aims at contributing to a better decision-making process by producing knowledge out of the data and information analyzed.

To give the insight of where our thesis is situated with respect to the big picture of BI, we introduce yet another definition given in [11], which is quite complete because it combines the 3-dimensions of BI:

"*The term "Business Intelligence", which first popped up in the late 1980s, encompasses a wide array of processes used to collect, analyze, and disseminate data, all in the interests of better decision making. Business intelligence tools allow em-*

*ployees to extract, transform, and load (or ETL, as people in the industry would say) data for analysis and then make those analyses available in reports, alerts, and scorecards.*"

And exactly the ETL part is where this thesis is situated. The reason is that our thesis is centralized on ETL processes, since we aim at analyzing them and the operations that characterize such processes and hence, provide a data generator that creates test data for evaluating ETL processes.

## 2.1 Business Processes

Business processes consist of an inter-related chain of activities and tasks that perform individually, but coordinate in a structured way with the purpose to reach a common desired goal.

### 2.1.1 BPM

"*Business Process Management (BPM) is the art and science of overseeing how work is performed in an organization to ensure consistent outcomes and to take advantage of improvement opportunities.*" [12].

BPM is a discipline that has evolved over time from general principles and methods, to more specific ones that aim at monitoring, analyzing and improving business processes in order to increase productivity and bring more value to the customers and company itself. BPM focuses on the manageability of business processes.

Based on the actor performing the activity, business processes can require the user interaction or can be automatically handled by information systems. Thus, business processes are an interesting topic for both business and computer science communities. The former one aims at optimizing the process and improving productivity and customer satisfaction, whereas the latter one aims at improving the underlying technology to accommodate faster and better support for the execution of complex business processes. To narrow the gap between these two communities and help them understand each other with the same language notations, BPMN comes into place by graphically representing business process workflows, meanwhile

enriching it with implementation details.

### 2.1.2 BPMN

Business Process Modeling Notation (BPMN) is a set of semantics and notations to graphically represent business processes workflows. BPMN is a widely used standard for business process modeling. Since generally business processes are executed in a specific order, representing them as a flowchart helps both technical and business users.

An advantage of BPMN is that it helps in graphically portraying also complex processes. Another benefit that comes with BPMN is the empowerment of business process workflow with execution details, thus providing a mapping between the graphics of the notation and the underlying constructs of execution languages such as Business Process Execution Language (BPEL).

## 2.2 Data Processes

A special type of business processes are data processes, which deal with the transfer and transformation of data. One key difference from other business processes is that data processes are almost fully automated and generally do not require human interaction. They are almost completely handled by the system. In this thesis we focus on ETL data processes.

### 2.2.1 ETL Processes

ETL, or widely known as Extract, Transform and Load, are processes that integrate information coming from disparate different systems into common repositories, which typically are Data Warehouses or Data Marts. ETL are data integration processes that help create the BI infrastructure for gathering and analyzing data, and finally turning them into useful knowledge and presenting in a suitable way i.e., reports, dashboards, graphics. ETL can be thought of as a flowchart of three main activities:

1. Extraction – the process of extracting that portion of data from the source systems that are useful for the strategic decision-making and required from the business analysts.

2. Transformation – the data obtained from the previous phase are then cleaned, transformed and integrated into a more representative way according to the company's business goals.

3. Loading – the integrated cleaned data are then loaded into big repositories that serve for analysis and reporting purposes. Typical repositories are Data Warehouses and Data Marts.

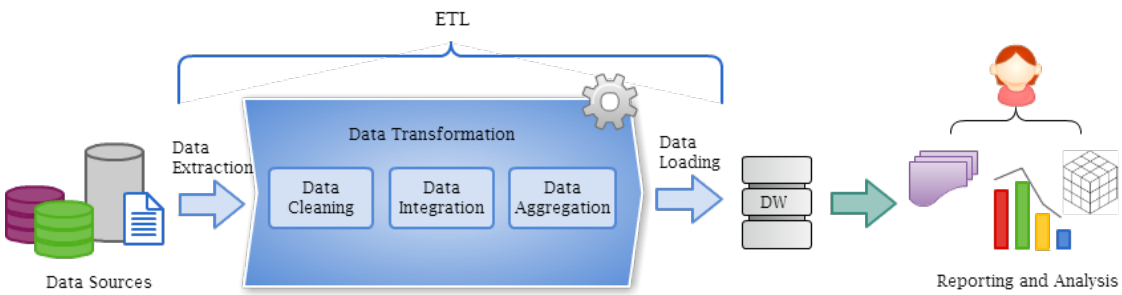



Figure 2.1: An ETL process

Figure 2.1 shows the three phases of an ETL process. It starts with the data extraction from the data sources, which might be of relational or non-relational structure. Then, the process follows with the data transformation phase, the most complex one that consists of three main steps: data cleaning, data integration and the data aggregation step. Lastly, the process ends with the data loading phase responsible for loading the cleaned, integrated and aggregated data into the Data Warehouse. Business analysts, use the integrated data of the Data Warehouse to carry analysis, build reports and transform the data into knowledge useful for strategic decision making.

Hence, ETL is a structured process with defined order of activities and execution details, where BPMN principles can be successfully applied for modeling it. As mentioned in the earlier section, ETL is a special type of business processes. It is a business process because it serves the business purpose to correctly integrate the

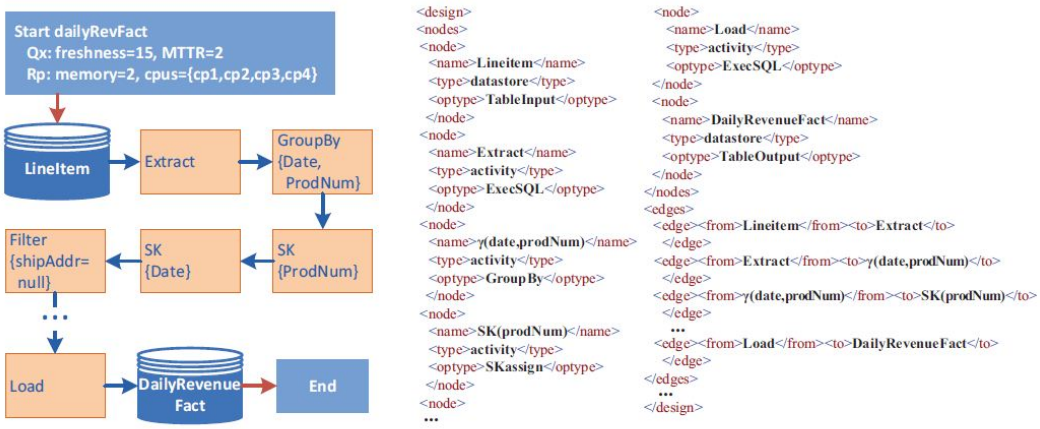
company's data, but on the other hand it is almost completely automated and does not require human intervention, and as such it is a data process. ETL deals solely with data activities and events rather than physical ones. There exists though a strong correlation of ETL, BPM and BPMN since same notations applied to managing and modeling business processes are also applied to modeling and analyzing ETL processes. This field of research has already been adressed by Zineb El Akkaoui, Esteban Zimanyi, Jose-Norberto Mazon, Juan Trujillo and Alejandro A. Vaisman in their studies [13], [14] and [2]. In the latter one, not only BPMN notations have been applied for modeling ETL workflows, but they have also been enriched with implementation specifics from BPEL concepts. More details about this correlation between ETL and BPMN will be presented in the upcoming section.

### 2.2.2 ETL Modeling

In the last years several modeling methodologies for ETL scenarios have been proposed, covering both conceptual and logical level. However, up to now there is no recognized standard and the current work address only a portion of ETL semantics. Usually, the modeling means are graphical notations that are vendor dependent and that can be executed in only one particular tool. Nevertheless, in the last decade there have been many efforts tackling the issue of defining a high level modeling framework for ETL that is platform independent. In addition, some of these efforts go beyond modeling the conceptual or logical level, by providing also an automatic generation of an executable code of the ETL model that can run in one particular ETL engine. Thus, these research studies abstract from tool specific ETL modeling and propose a logical ETL design that attempts to generalize and accommodate all ETL process. Their main contribution is the representation of this layer in a formalized language like XML that makes it easy to further analyze.

In [15], [16], [17], the authors propose unified modeling of ETL processes. The main contribution of this approach is introducing a generic metamodel layer that is independent of any specific tool. ETL scenarios are represented by means of a graphical interface and declarative languages. ETL flow is modeled at the logical level as a directed acyclic graph consisting of two main graphical notations: nodes

that model ETL entities (activities, data stores and corresponding schemata) and edges that model the relationship between them (the data flow from the source to the target entity). Attributes are treated as "*first class citizens*" and data transformations are captured in details. The graph modeling is very expressive but can get quite complex when many sources, activities and attributes are present in the model. However, they provide the means to reduce the complexity by adding the *Graph Transformation* functionality that allows capturing only the high level view of the model.




(a) ETL graph (b) xLM representation

Figure 2.2: Logical modeling layer of ETL flow. Example taken from [1]

.

In [1], a more complete ETL modeling framework is presented, compared to the previous studies. The reason is that authors introduce a layered ETL design approach by modeling also the conceptual view of the ETL apart from the logical one, meanwhile enriching it with the optimization objectives called QoX. Business process models are used for designing a unified conceptual model. Whereas, for the logical design they propose an XML representation and they call such model *xLM*, which is independent of a specific ETL platform. Using business process models offers several advantages to the framework. Apart from being a wide-spread standard, it enables an ETL model expressed in business terms by hiding the technical details to business users. As for the logical modeling, the same graph principles are followed as in the previous work, but enriched with additional parameters in order to incorporate business requirements and the QoX objectives. Another contribution

of such work apart from the layered approach is the automatic translation from one layer to the other and the generation of the physical model which is specific to a particular ETL engine. In a sequel work [18], this approach has been extended by focusing on the optimization objectives of ETL flows. They bring into attention the need to consider both functional and non-functional requirements while dealing with the layered design of the ETL process. They include such optimization objectives into the conceptual, logical and physical modeling. This work continues in [19], where they propose a framework which produces a physical design that is optimized according to the QoX objectives. The benefit of such approach is the optimization framework that allows the flow to execute in that engine, which is the most optimized according to those quality objectives. This framework is complementary to the previous work in graph modeling [1] and to the *Optimizer* component of the *xPAD* cross-engine platform presented in [20]. The quality objectives are captured at the early stages of collecting user requirements, and are presented as properties of the flow in the xLM representation of the logical model.

Work in [13] proposes a framework for model-driven development of ETL processes which allows creating a platform independent conceptual model expressed in terms of *BPMN4ETL* metamodel. BPMN4ETL is a platform independent design model presented in this paper that extends standard BPMN notations with additional constructs to accommodate ETL design. Additionally, in [2] the authors propose automatic generation of executable code for running an ETL process on a specific platform. The framework tackles the design and implementation phase of ETL process development. However, this methodology is still not validated for performance, usability or flexibility issues. A similar line of work is presented in [14] where a BPMN-based metamodel is considered for generating conceptual models of ETL. But, in addition to the previous approaches it captures both the control and data flow of the process by representing also the finer granularity of how input data are transformed to produce the output data.

The above mentioned research works are similar in spirit because both focus on representing the conceptual view of ETL by means of BPMN notations which is a widely accepted standard. Additionally, both provide vendor dependent code

generation for running the ETL process in a chosen tool.

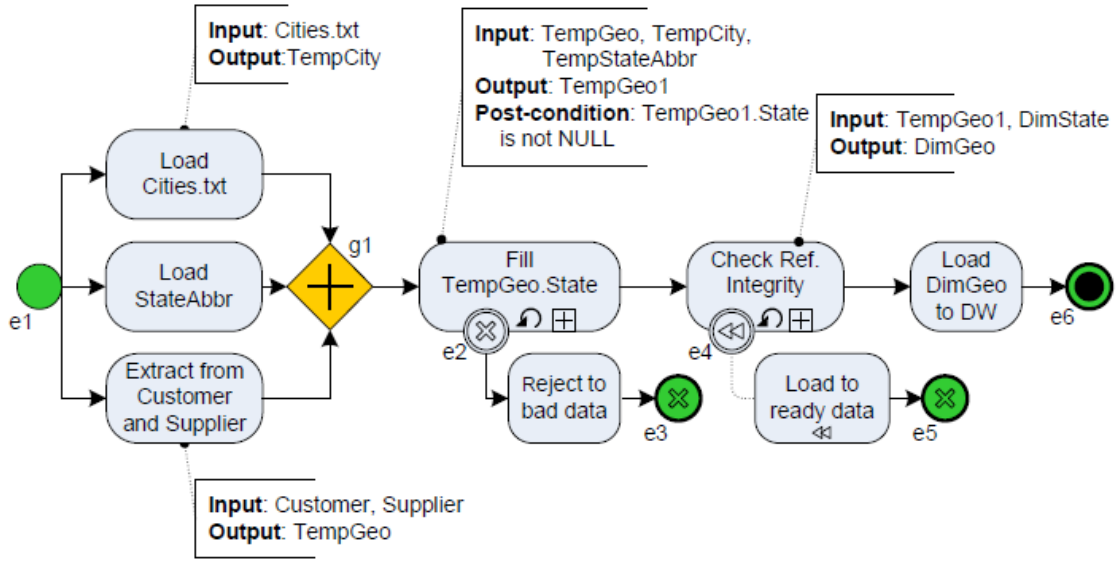



Figure 2.3: ETL flow modeled in BPMN. Example taken from [2]

.

However, in [1] they extend the work by modeling also the logical level through xLM language (similar to XML) and transforming it into the physical one, while considering also optimizations criteria during the entire procedure. The use of xLM to encode the logical model is closely related to the fact that most of the current ETL tools already use XML to encode their ETL designs. While in [13], [2], [14] they focus mainly on the conceptual and physical level. The main advantage of having a logical design is to create an expressive uniform formalism for representing ETL models independently from the tool they are designed or executed.

Another modeling approach has been presented in [21], that uses UML for the conceptual modeling of ETL processes. They rely on this well-known modeling standard and its simplicity for representing the ETL tasks and their relationships. UML packages are being used for modeling complex ETL scenarios and for designing the global view of the processes. This is a different line of approach in contrast to [15], [16], [17] in which attributes are considered "first class citizens". However, the following work [22] from the same authors proposes modeling ETL processes at different levels of granularity and thus capturing also the data mappings and relationships between attributes. They propose the use of extended UML packages

to represent relationships between source and target systems at different levels of details, thus enabling modeling at the conceptual, logical as well as at the physical level. This is a rather novel approach that uses the same notations (UML) to accomplish the conceptual, logical and physical design. One major contribution is providing expressive ETL designs by extending the formalisms of UML in order to capture data mappings at attributes level.

Another line of work is presented in [23], [24]. It focuses on the semi-automatic design of the conceptual modeling of ETL by using an ontology-driven approach. The conceptual layer is represented as a graph, while data mappings are generated step by step based on graph transformation rules derived by the semantics attached to the domain ontology. Considering that the ontology used is based on a specific domain related to the type of information residing in that particular data warehouse, this creates some limitation in the generalization of the ETL processes.

In this research study, we look at the ETL process from the logical design perspective because we need to extract and analyze the schema mappings and attributes relationship. For this purpose, we first need a generalization of ETL semantics and then a simple yet expressive representation of data mappings in order to be able to analyze it and extract relationships between ETL activities and operations. Based on these requirements, the layered modeling framework presented in [1] can be useful for our scope, since it offers high level of expressiveness with respect to data mappings and attributes relationships in a very structured way. While, UML approach lacks in the expressiveness power of the model. It does not provide a structured generalized form of representing data mappings, which are rather defined by means of natural language syntax. Ontology approach too is not detailed enough for our purpose, because it provides only the conceptual modeling and not the logical one. The BPMN metamodels are expressive and show the ability to capture the input data transformations in a quite detailed way, but they do not offer the means to provide a common representation language. They offer only the possibility to transform it in XML based representation for BPEL for instance, which is specific to a dedicated tool. Another reason we choose [1] is the automatic transformation of the business process representation into xLM, which represents the logical model, and thus en-

abling to represent in a very comprehensible and expressive manner most of the ETL semantics. One major contribution is also capturing non-functional requirements or QoX objectives. Such optimization objectives are represented as properties of the flow in the xLM representation. Given this representation we can extract both functional requirements (rules, data mappings) as well as non-functional requirements related to optimization criteria, and recreate a sample of input data that satisfy the rules and conditions extracted from the given ETL process.

## 2.3 Quantitative and Qualitative Analysis of Business Processes

Business environment is very dynamic and is pushing towards rapidly changing business processes in order to adapt to the continuous changing environment they operate in and to the evolving business needs. Along with this, comes the necessity for actively reacting to the new challenges and faster and better decision making. Therefore, the need for real-time Data Warehouse solutions is a main concern in Business Intelligence. This calls for real-time data processes in order to reflect instantly the dynamic business environment. Nevertheless, often these changes in business requirements can cause business processes to become inefficient. Moreover, in order to adapt to the changing requirements, business processes need to be redesigned or remodeled effectively and efficiently. Tracking these redesign changes and the impact they have on performance is very important to ensure the reliability and high quality of the process. For this reason, it is required a continuous analysis, qualitative and quantitative, of the performance and flexibility of business processes.

Performance can be defined as the degree to which a system or a process satisfies the objectives for which it is aimed for. Performance can span in many dimensions, depending on user requirements as well as on the particularities of the process type under analysis. Referring to [12], typical performance metrics that can be applied to any process are: time, cost, quality and flexibility. The reason is that any company urges for better, faster and cheaper processes. Hence, these are the four most typical and general performance measures applied to any business process case. Anyhow,

there are many other specific metrics that deal with the particularities of each process. This is the case of ETL process, where other specific quality criteria can be defined and analyzed. ETL is a critical process whose performance is of high importance to decision making at the managerial level. Thus, analysts should pay special attention to all the quality dimensions that concern such a process. For instance, just to name a few, other important performance requirements are: latency, throughput, utilization, capacity etc. Broader information is given in the detailed taxonomy presented in Figure 2.4, extracted from [3].

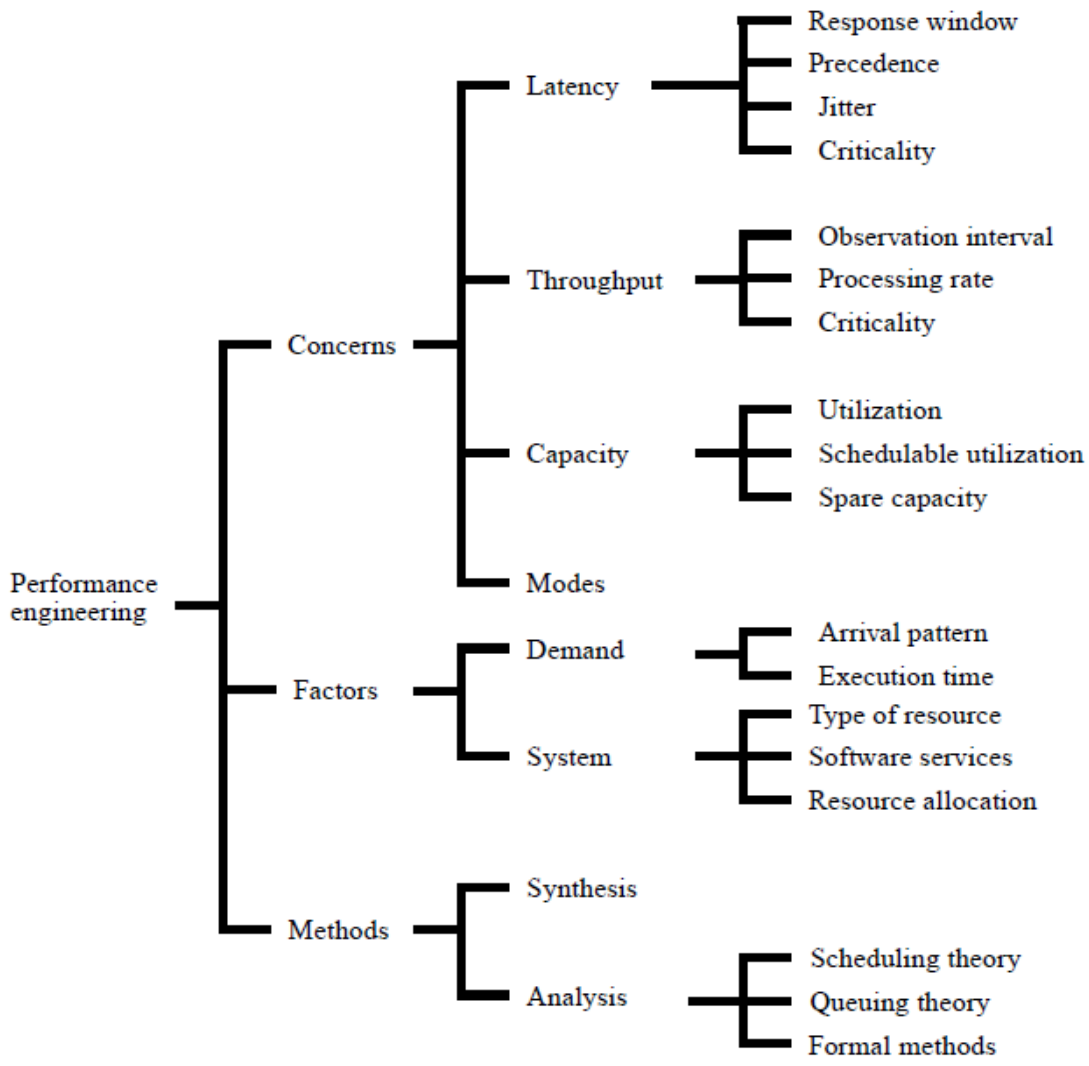



Figure 2.4: Performance taxonomy taken from [3]

Qualitative analyses aim at identifying the weakest parts of the process which cause delays, inefficiencies or are redundant to the overall purpose of the process.

Many qualitative analysis techniques exist, and two well-known are listed below:

- Value-Added Analysis – Aims at identifying the tasks that are redundant to the overall business process and bring no value to the end user, which might be the customer or the business itself. The inefficiencies are typically related to time and delays, due to unnecessary steps performed. Hence, such technique tries to identify and eliminate these unnecessary tasks (waste elimination) and thus reduce costs.

- Root Cause Analysis – Is another set of technique broadly used in discovering and identifying the reasons of unwanted behavior and inefficiencies in the business process or production line. Cause-effect analysis and Why-why diagrams are two representative methods that focus on identifying the reasons of undesired outcomes or overall inefficiencies.

However, finding just the causes that bring to low performance is not enough. Sometimes the insights acquired from the qualitative analysis are not sufficiently detailed. Hence, being able to quantify the severity of the problem is a better option for the purpose of continuous improvement of the process. In order to measure the performance objectives analysts apply quantitative analyses. They consist of a set of principles and techniques for evaluating and measuring the degree to which the process under study satisfies the performance requirements. Such quantification of process quality is important for comparative analysis as well as decision making.

As mentioned previously, there are many performance metrics that can be addressed during the quantitative analyses, but the most typical ones focus on time, cost, quality and flexibility. Nevertheless, it is important to realize that there are many trade-offs between conflicting quality criteria and hence satisfying all of them is impossible. Rather, it would be optimal to find that solution that on average tends to satisfy all quality dimensions that cope with user requirements. Quality objectives might address the entire process as a whole entity, or parts of it, this depends on the specific user needs.

Many quantitative techniques measure the performance objectives for the complete process by starting from the performance of individual activities and resources

in that process [12]. One advantage that they offer is that by providing this finer granularity of the process, analysts are able to "easily" measure changing processes by keeping track of the changed resources or activities. However, for complex process models with complicated dependencies amongst activities, it becomes difficult to accurately measure the performance of the complete process. Some representative quantitative analysis techniques are:

- Flow analysis - Aims at calculating the performance of the overall process by estimating the performance of individual activities first. One main assumption is that the performance measures (i.e., time, cost, flexibility or quality) of each activity are available. Is easy and mathematical calculations are intuitive when dealing with simple process patterns, but can get complex as the complexity of process flows arises. A drawback of the flow analysis is that it does not consider the variation in performance of each activity due to variable workload. Moreover, it is not always applicable, for example in the presence of multiple overlapping cycles.

- Queuing theory - Another set of mathematical techniques that aims at calculating the performance by taking into account the resource contention, unlike the previous method. Since variations in workload create queues and waiting time, the queuing theory tries to estimate queues parameters in order to analyze the overall system performance by considering one activity at a time. Similarly to the flow analysis, the mathematical computations can get quite complex especially when dealing with concurrent activities.

- Process simulation - Is a wide-spread technique that models and simulates a real business process for analysis purposes, typically what-if analysis. After each simulation run, execution details are collected and then further analyzed in order to compute the performance metrics mentioned above such as: total execution time, utilization rate of specific activities, capacity and many more. In the following chapter we dedicate a complete section to business process simulator engines that accomplish this task.

# 3 RELATED WORK

In this chapter we discuss the literature and research study on ETL tools that we examined during our research. Then, we proceed with the literature review of ETL benchmarking, simulation and evaluation. Lastly, we talk about current data generation tools already in the market, and analyze the possibility of adapting them to our specific purpose of generating data for ETL processes.

## 3.1 ETL Tools

Many tools have been developed to support the modeling and execution of ETL processes. Generally, they are known as data integration tools. In order for a software to be categorized as a data integration tool, it should provide some specific capabilities. Gartner Group [25], provides a framework of the basic criteria that a software should possess in order to be qualified as a data integration tool. These criteria include the following features:

- The ability to interact with a range of different types of data structures, e.g., relational DBMS products, flat files etc.
- Data delivery capabilities in a variety of modes such as bulk/batch mode, federated views etc.
- Basic data transformation capabilities such as: data type conversions, string manipulations and calculations.
- Metadata and data modeling support.

- Deployment options and runtime platform capabilities i.e., Windows, Unix and/or Linux.
- Data governance support capabilities e.g., data profiling, cleaning and mining features.
- Operations and administration capabilities includes facilities for enabling adequate ongoing support, management, monitoring and control of the data integration processes.
- Support for SOA deployments: the ability to deploy all aspects of runtime functionality as data services, interaction with service repositories and registries etc.

Some of the leading data integration providers are: IBM[1], Informatica[2], Oracle[3], Microsoft[4], Talend[5], Pentaho[6], Information Builders[7], etc.

There are many research papers that provide a comparative analysis of the market leading ETL tools, such as [25] and [26]. They analyze in depth the functionalities and capabilities that these tools offer, and from that can be derived that all of them provide support to all features that define data integration tools. But, they do not offer any support for data generation neither optimization of the flow.

## 3.2 ETL Evaluation and Optimization

As of [27], optimization techniques are usually done in an ad-hoc fashion based on the experience of the designer. Moreover, the only optimization carried out is from the optimizer of the DBMS during the loading phase. From the studies conducted

[1] http://www.ibm.com
[2] http://www.informatica.com
[3] http://www.oracle.com/index.html
[4] http://www.microsoft.com
[5] https://www.talend.com
[6] http://www.pentaho.com
[7] http://www.informationbuilders.com

around this area, there have been attempts to optimize the flow, but again this was manual and responsibility of the designer.

To tackle the issue of ETL flow optimization, there are many studies conducted that throw light on techniques and methods that can be followed in order to optimize the ETL process. This lack of support from the ETL vendors is due to the fact that these optimization approaches basically are manual and require human interaction. Meanwhile, ETL flow is almost fully automated apart from the design phase and maybe some extra data cleaning procedures done by the end user in order to ensure high data quality. Another limitation comes from the "black-box" nature of several ETL activities whose semantics are unknown to the designer, hence the optimization problem becomes difficult. To this end, in this section we talk about logical ETL optimization approaches by discussing related literature on the topic.

As it is argued in [18] optimizing ETL flows solely for performance objectives is not enough. ETL flows are very complex and other metrics e.g., cost, recoverability, maintenance, latency, freshness etc., referred to as QoX need to be considered during the design phase. Current ETL tools do not capture such quality objectives and neither do they provide a formal mechanism to quantify, track, and measure them [18]. In this paper Simitsis et al., present a framework of including multiple optimization objectives during the design phase of ETL flows. They propose several techniques to optimize the flow for each individual objective. For example, to improve performance it is advised to make use of algebraic optimization rules that can be applied to the ETL design phase as well, for instance pushing selections, and in general most restrictive operations at the beginning of the flow. Other optimization techniques for each QoX objective are presented. Their contribution goes beyond by providing a framework of ETL design by considering multiple QoX and also trade-offs among them.

## 3.3 ETL Process Benchmarking

Apart from optimizing ETL workflows, there is a vast amount of work also dedicated to benchmarking ETL processes. This necessity comes from the differences present

in the distinct design approaches followed by each ETL tool and each designer. Moreover, both researchers and industry are particularly interested in benchmarking ETL processes in order to compare and evaluate products and prototypes. Just to name a few, some prominent works in this area are [28] and [29]. In both these studies it is noted the lack of a widely accepted standard for evaluating integration processes.

In the first one, [28], the benchmarking attempt focuses on providing a taxonomy of frequent design cases of ETL workflows. In addition, they also provide the basic configuration parameters and measures to be considered when assessing benchmarking goals. As such, they distinguish several design patterns: Line, Primary Flow, Butterfly, Fork, Tree etc., and also these parameters: size of workflow, size of input data, latency, workflow selectivity, completion time etc. This ETL workflow benchmark has a double importance, first it helps during the design phase by decomposing also complex processes into a combination of the above constructs, and second it can be used for optimization purposes too. As they state, "*The main message from our work is the need for a commonly agreed benchmark that reflects real-world ETL scenarios, both for research purposes and, ultimately, for the comparison of ETL tools.*"

The first work focuses on defining a benchmark at the logical level of data integration processes, meanwhile assessing several optimization criteria called configuration parameters. Whereas, the other work [29], focuses mostly on the physical level by providing a multi-layered benchmarking platform called *DIPBench*, used for evaluating the performance of data integration systems.

Despite these and many other attempts to bring a standard into the process of ETL, still no agreement has been reached yet. An important factor in benchmarking is defining similar workloads while testing different ETL scenarios, that is why with this thesis we aim at generating test data for later evaluating ETL flows and measuring QoX objectives.

## 3.4 Data Generators for Benchmarking Purposes

This section is devoted to the study of data generator frameworks that already exist. Specifically, we study the approach they follow for simulating data sets manifesting real-world characteristics. As introduced in Chapter 1, our goal is to generate synthetic input data for the ETL process.

Many research areas require large sizes of data to work with. However, acquiring huge data sets is not often a feasible solution. First of all, transferring over the network these large amounts of data is expensive. Secondly, quite often there are faced confidentiality issues that do not allow organizations to make their data public for research and study purposes. Consequently, many research teams are working on the design and implementation of data generators that are able to simulate real-world data sets for the purpose of benchmarking and analysis. As a result, this field has captured attention worldwide, and ours too.

[30] Presents one of the first attempts to generating synthetic data, which is later used as input for workloads when testing the performance of database systems. They concentrate on the algorithm characteristics of how to scale up and speed up the data generation process using parallel computer architectures.

In [31], is presented a tool called Big Data Generator Suite (BDGS) for generating Big Data, meanwhile preserving the 4V characteristics of Big Data i.e., volume, variety, velocity and veracity. This tool is used to generate textual data, graph and also table structured data sets. As of [32], BDGS is part of BigDataBench, a data benchmark suite of six real-world datasets and nineteen big data workloads, covering six application scenarios: micro benchmarks, Cloud "OLTP", relational query, search engine, social networks, and e-commerce. The data generation process goes through four steps. The first step is data selection, that aims at collecting representative real-world data. The second step is data processing during which important characteristics are extracted from the data sets, e.g., in textual data sets this is achieved by applying topic models in order to capture the idea conveyed by that particular data set. Meanwhile, for the graph generator they apply Kronecker graph model that enables capturing relevant graph patterns and finally generating a self-similar graph. The next step is generating the data based on the characteristics

acquired from the previous step. For the table structured data generator they use PDGF, which is a parallel data generation framework suitable for cloud scale data generation.

PDGF is broadly presented in [33]. This tool is platform independent. It is tested for both Windows and Linux operating systems. PDGF generator uses XML configuration files for data description and distribution. In addition, it is responsible for the structured data generation part of the BigBench data model presented in [34]. The current BigBench schema is built on top of the TPC-DS schema. It addresses systems such as DBMS and MapReduce. This tool follows a similar approach to ours by generating data sets in a table structure fashion, starting from an XML representation. One advantage it offers is the ability to generate data sets that are correlated to each other and also based on statistical distribution functions. Moreover, it supports additional plug-ins to accommodate specific needs.

Another data generator tool is LinkBench introduced in [35]. It is a database benchmark that offers real-world database workloads for social applications, especially Facebook. It is limited to graph data sets and currently works only with MySQL database system. It offers good capabilities for simulating social networks, and generating different workloads by considering data access patterns, system resource utilization etc., but is not relevant to our ETL specific purpose. Considering that LinkBench generates graph workloads (graph contains nodes and edges), it means that can generate data which can only comply to two possible schemata: schema of the node and schema of the edge. Hence, it does not capture schema changes from one node to the other, as it is the case of the graph representation of the ETL flow. Meanwhile, our solution's objective is exactly to capture flow semantics and schema changes, thus this tool does not provide those functionalities we require for our initiative.

HiBench introduced in [36], is a benchmarking suite specifically designed for Hadoop. Its aim is to quantitatively evaluate the performance of Hadoop framework. The added value of this benchmarking suite is the fact that it considers not only synthetic micro-benchmarks, but also realistic workloads coming from complex data analysis Hadoop applications. HiBench is not a data generator itself but rather a

performance evaluator of Hadoop, based on Hadoop applications workloads. Hence, being Hadoop oriented, it is not generic and does not fit with our need to generate ETL workloads.

[37], [38], [39], show three prototype tools that generate synthetic data sets based on XML representation of the data layout. In [37], a parallel synthetic data generator (PSDG) is presented. It aims to generate across multiple processors, realistic industrial data sets that follow the characteristics of real data. Similarly to PDGF, [33], it is based on a description language (XML) for the definition of data layout. It is quite limited in the generation capabilities considering only few generation constraints such as min/max, distribution, formula, iteration etc. Another similar data generator tool is the multi-dimensional data generator (MUDD) presented in [38]. It supports the generation of synthetic data sets by applying statistical distribution functions, as well as real-world data sets by using existing realistic dictionaries. Another prototype tool for generating synthetic data is presented in [39]. Called Information Discovery and Analysis Systems Data and Scenario Generator, shortly (IDSG), is developed for generating workloads for testing and training purposes of data mining tools. Similarly to the above mentioned tools, they separate the structure of the data to be generated and the specificities of data generation engine by using XML representations. One additional feature it offers is the semantic graph representation at the conceptual level of the attributes relationships. All tools allow for inter-rows, intra-rows and inter-table dependencies when generating data sets. However, this is achieved by an iterative approach starting with generating the independent data first and then proceeding to the other data which are correlated to the previously generated ones. Consequently, they do not provide independent generation of dependent data sets as opposed to PDGF, [33], which offers the capability of generating data with cyclic dependencies. The provided functionality of generating dependent datasets is crucial for our solution. The reason is that ETL performs complex transforms over the extracted data in order to produce aggregated information, which is hence derived from the crossing of extracted data. A simple example would be an ETL process that crosses customer data, which are distributed over many data sources, and finally producing several customer views by

aggregating them by customer's personal criteria i.e., age, birthplace etc., or other customer's product criteria i.e., loan, mortgage amounts etc. This obviously requires data crossing and transformations in order to derive the final views. For our scope, it means that we need to be able to understand these complex transformations and generate input data that simulate the flow entirely, such that when replaying the flow they can successfully derive the exact information that is intended to by the ETL process.

Another data generator based on description language is introduced in [40]. It presents a functional language called Data Generation Language (DGL), that enables the generation of databases with inter-table dependencies and of complex distribution nature. The program written in DGL can be compiled into C++ code. Alternatively, a thin layer on top of DGL allows to extend the SQL create table clause with the DGL additional data generation features. DGL is a functional language that offers limited capabilities in parsing an XML representation of a DAG input and then parsing, analyzing its semantics and finally generating data based on the parsed expressions. In addition, it is limited only to the generation of workloads for relational DBMS systems.

To sum up all our findings regarding the data generator tools, we have structured the information provided above in an illustrative table. Since we are interested in generating table-structured data, we considered only those tools that provide this capability.

| **Features** | **Data generator tool** | | | | |
|---|---|---|---|---|---|
| | **PDGF** | **PSDG** | **MUDD** | **IDSG** | **DGL** |
| No Reference Generation | – | – | – | – | – |
| Scanning References | – | + | + | + | + |
| Computing References | + | – | – | – | – |
| XML-based | + | + | + | + | – |
| Workload of dynamic schema | – | – | – | – | – |

Table 3.1: Data generator tools' feature support

We have listed the tools' short names and the important generation features for our approach. As already mentioned, a basic feature is the generation procedure of references, since ETL consists of complex transformations that produce aggregated data derived from the ones extracted from the source systems. There are three scenarios that are followed:

- No Reference generation – These tools do not consider relationships between tables. Hence, correlated tables cannot be generated. For example the constraint of primary key – foreign key dependency cannot be guaranteed by these tools.

- Scanning references – Generating dependent tables requires to simultaneously scan all references of referenced tables. Firstly, tables containing independent data are generated. Then, the dependent tables are generated as a derivation of related tables.

- Computing references – Supports the generation of cyclic dependencies since dependent tables can be generated independently. Unlike the previous case, there is no need to wait first for the generation of related data, and then proceed to deriving the dependent data. Instead, both processes are executed independently by performing powerful calculations. This is a strong technique used for the parallel data generation.

In addition, we considered also other characteristics that are crucial to the ETL data generator solution:

- XML-based – Reliance on a description language i.e., XML. This is due to the fact that we base our work on the logical modelling of ETL as a directed acyclic graph (DAG), discussed in [1], and formalized by an XML representation.

- Workload of dynamic schema – Presents the ability to identify and track schema changes and finally generate workloads of dynamic schema. None of the above analyzed tools support this feature, since they are all tailored for DBMS workload generation and hence, the schema of the generated workload

is always static. While, for our purpose we need to generate workloads for many ETL scenarios that have different schemata. Therefore, we strongly require the capability to identify the source schema and track dynamically the changes throughout the flow.

To show the support of a given feature we use the plus (+) sign. Otherwise, we represent with an hyphen sign (–) the lack of such capability.

All the above mentioned tools provide powerful capabilities to address the issue of generating data for testing and benchmarking purposes for DBMS. As such, they are not particularly tailored for ETL data and are not capable of generating workloads of dynamic schemata, rather they generate data for a particular set of source schemata. Hence, they lack the capability to scan and analyze ETL operations mapping rules, constraints and track schema changes. Therefore, we decided to design our own framework of ETL data generators, specifically tailored for ETL processes.

## 3.5 Constraint-based Data Generation

Study in [41] proposes a tool for populating the database with meaningful data that satisfy database constraints. They introduce a semi-automatic approach, which focuses on the correctness of DB systems and additionally they restrict attention to relational databases. This work continues in a sequel, [42] where they introduce a framework that offers the approach presented in the previous work. They offer database population with meaningful data that satisfy database constraints. These constraints are expressed in SQL and parsed by an SQL parser, able to parse the semantics of database schema and constraints. It is a semi-automatic tool, which includes an automatic parser, but requires user interaction when generating the tables. However, their framework is restricted to relational databases semantics only, and particularly deals with correctness of DBMS, not quality. In addition, it is not fully automated and is tied to SQL language. Hence, it does not provide data generation based on other semantics apart from SQL semantics tailored for database systems and cannot be adapted to our data-centric data generation needs.

In [43], it is presented a fault-based approach to the generation of database in-

stances for application programs, specifically aiming to the data generation problem in support of white-box testing[8] of embedded SQL programs. Given SQL statements, the database schema definition and tester requirements, they generate a set of constraints which can be given to existing constraints solvers. If they are satisfiable, desired database instances are obtained.

Work in [44] aims at introducing efficient algorithms for generating synthetic databases that satisfy a given set of cardinality constraints. Their approach differs from the above mentioned studies since they do not generate a database instance by modifying an existing one, but rather their data generation algorithms take only the constraints as input (even though the constraints may be extracted from existing databases).

In [45] they propose a multi-objective test set creation. They tackle the problem of generating branch adequate test sets, which aim at creating test sets to guarantee the execution of each of the reachable branches of the program. Their innovation though, is that they formulate the problem as a multi-objective search problem focusing not only on covering the branch execution, but also on additional goals the tester might require i.e., memory consumption criterion.

In [46], they propose a query-aware test database generator called QAGen. The generated database satisfies not only constraints of database schemata, table semantics, but also the query along with the set of user-defined constraints on each query operator.

This paper, [47], presents a generic relational data generation tool specific to database systems. They built their data generation approach on top of a graph model, which as they argue, makes it easy to generate data also for complex database schema with many dependencies (i.e., inter and intra table relationships). The proposed tool provides an extensible data generation based on cardinality requirements (primary, unique, and foreign-key constraints) and other customizable parameters such as data types, type of output, functions, and distributions. They support test data creation based on several attribute properties: datatype, load size (of the input

[8]White-box testing is a software testing method that focuses on testing internal structures of an application.

datasets) that can be parameterized. They also handle dependencies (i.e., foreign key-primary key dependency, intra-row, intra-column and intra-table dependencies) by following the approach to first generate non-referenced data and then the referenced ones, based on what was previously generated.

However, all the above bodies of work focus only on relational data generation by resolving the constraints of the existing database system. Our scope is similar but broader, given that we do not restrict it to only relational schema and we tackle different types of constraints not represented by SQL semantics. In addition, we do not depend on a single database instance schema, rather the datasets we generate vary based on the input type and schema of the data process considered.

## 3.6 BPMN Simulators

Simulation is a technique that imitates the behavior of real-life processes or systems. It is widely used to predict and evaluate how the process is performing under certain circumstances. This comparative analysis of process behavior is important for understanding and redesigning processes. Simulation models are typically used for what-if analysis in order to compare the redesign impact in the process performance. By executing the simulated process model many times, it can be determined the performance baseline of the process for each particular simulation scenario. Simulation techniques can be successfully applied in predicting the process performance before its actual implementation as well as during its lifetime and hence helping in the continuous improvements. Simulation provides quantitative measures that are very helpful for decision-making during the re-engineering phase and also for understanding the process behavior and reaction due to changes in the process flow.

To accomplish simulation analysis there are many Business Process Management (BPM) tools that offer simulation capabilities. This section is devoted to the study of such tools. Although each tool is particular, the process simulation workflow as introduced in [48], includes the following steps:

1. Define modeling objectives – It should be clear which is the outcome expected and the measure that is going to be analyzed at the end.

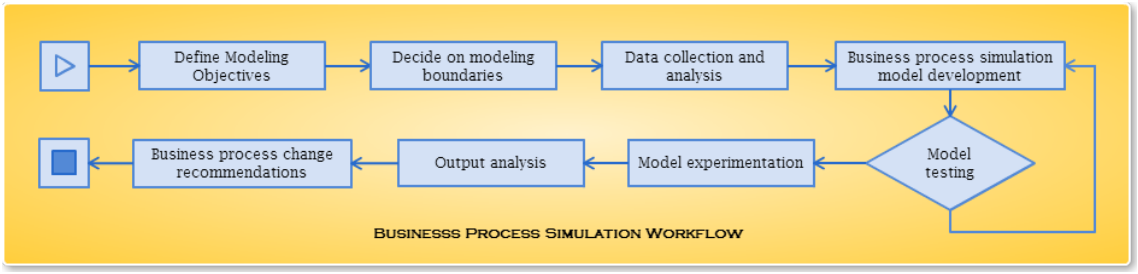



Figure 3.1: BP simulation workflow

2. Decide on modeling boundaries – Modeling processes in details is not possible, so it is necessary to include only the most relevant and critical parts of the process in the representative model.

3. Data collection and analysis – Based on the level of detail represented in the model and on the modeling objectives, data has to be collected and further analyzed via statistical tools in order to be parameterized through stochastic functions and given as input during the simulation.

4. Business process simulation model development – By using BPS tools the simulation model is constructed on the required level of detail.

5. Model testing – An iterative step carried during the simulation model development in order to assure the accuracy and precision of the built model.

6. Model experimentation – Consists of the simulation runs and results gathering.

7. Output analysis – Results collected from the previous step are analyzed using statistical techniques.

8. Business process change recommendations – Conclusions derived from the simulation output analysis are used for decision-making purposes, re-engineering and re-design recommendations in order to improve the performance of the process.

In order to compare the adaptability of current software tools for simulation purposes, there have been proposed several criteria for the modeling and simulation functionality that a tool should dispose. In [49], they mention main quality criteria for the modeling part of business processes, which are explicitly listed below:

- Correctness, the model needs to be syntactically and semantically correct.
- Relevance, the model should not contain irrelevant details.
- Economic efficiency, the model should serve a particular purpose that outweighs the cost of modelling.
- Clarity, the model should be (intuitively) understandable by the reader.
- Comparability, the models should be based on the same modelling conventions within and between models.
- Systematic design, the model should have well-defined interfaces to other types of models such as organizational charts and data models.

Whereas, for the simulation capabilities main requirements identified as per [50] are:

- General capabilities, including modeling flexibility and ease of use.
- Hardware and software considerations.
- Animation, including default animation, library of standard icons, controllable speed of animation, and zoom in and out.
- Statistical capabilities, including random number generator, probability distributions, independent runs (or replications), determination of warm up period, and specification of performance measures.
- Customer support and documentation.
- Output reports and plots, including standard reports for the estimated performance measures, customization of reports, presentation of average, minimum and maximum values and standard deviation, storage and export of the results, and a variety of (static) graphics like histograms, time plots, and pie charts.

A comparative analysis of most well-known general purpose software modeling tools is presented in [51]. Tools under study are: Protos, ARIS, FLOWer, FileNet, Arena and CPN Tools and the criteria considered are the above mentioned ones, divided into three main categories: modeling, simulation and output capabilities.

- The modeling capabilities criteria evaluate the ease of modeling and most importantly the correctness and accuracy of the model built against the real process.
- The aim of simulation capabilities criteria is to evaluate how the simulation is being conducted and which options of simulation scenarios are parameterized and along which performance dimensions.
- The purpose of output analysis criteria as suggested by the name is to evaluate the output of a simulation process; Specifically, which portion of data are available to analyze, what kind of analysis can be carried and how are they presented to the final user.

The conclusions reached from the study of the above body of work show that most of the current BPM tools do not offer at all simulation capabilities (Flower) or at least they do not offer simulation with stochastic parameters or statistical analysis (FileNet). Whereas, the other tools analyzed in the same study: Arena, ARIS and CPN Tools, show to be qualified for process modeling and simulation. The reason is that they give support for the three main criteria mentioned above and considered to be fundamental for process simulation tools. However, they provide the simulation of business processes rather then data processes. As stated in the tools survey [51], their generated output is a quantitative analysis, time-based and cost-related information about process execution and resource utilization, rather than the data itself passing through the process. As a conclusion, BPS tools do not generate workloads, rather than comparative analysis of the execution details.

# 4 FORMALIZING THE PROBLEM OF DATA GENERATION

This thesis aims at providing an approach to generate tests data for ETL workflows. By analyzing the data flow transformation semantics, we provide the means to automatically generate representative input data for data processes that can successfully replay the flow.

In this chapter we discuss the pre-requisites of the data generation process. In order to accomplish the objectives defined in the first chapter, we came up with the following list of requirements for our framework prior to implementing the generation algorithm:

- Define the List of ETL Operations

  In order to develop a generic framework for ETL data generation, we had to first define a complete list of ETL operations, according to our study of popular integration tools and related literature. For this purpose, pioneer vendors that have been long in the market were considered such as: SSIS and Oracle Warehouse Builder and two newer but popular tools: Pentaho Data Integration (Kettle) and Talend.

- Categorize the List of ETL Operations

  The final list defined from the previous step, needed to be categorized based on those properties that are helpful for our data generation methodology.

- Formalize Semantics of ETL Operations

Our main source of information for the generation process are the semantics of ETL operations. Usually, operations have a rather complex set of semantics. Thus, it is necessary to capture them in a simplistic, but yet expressive way that would help us in later analyzing them during the data generation process. Therefore, we analyze ETL operations' transformation semantics and finally represent them by symbols and notations of the first order logic. These semantics generally consist of: cardinality requirements, rules, constraints and logical predicates that are evaluated during the operation, schema transformations.

Lastly, after analyzing each operation separately, we proceed to the analysis of the entire flow as a whole. Based on the knowledge extracted from the ETL flow and ETL operations, we are able to generate data that satisfy all the rules and constraints extracted.

To accomplish the above mentioned tasks, this work is largely based on the ETL taxonomy proposed by [4] and on the semantic-aware data generator presented in [52].

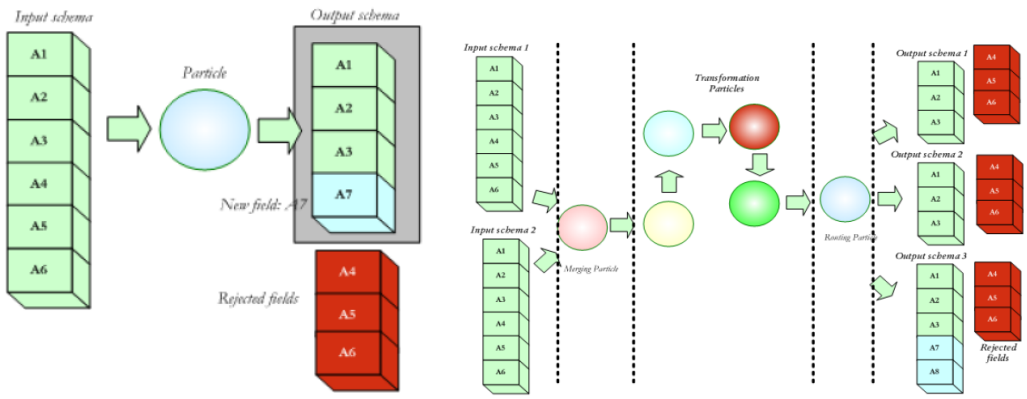
## 4.1 Defining the ETL Operation

First of all, before proceeding to the categorization of ETL operations discussed thoroughly in the following section, we need to present the notion of the ETL operation itself.

*We consider an ETL operation every activity of the ETL flow that applies transformation logic on the input dataset(s).*

This term is independent of the number of input and output datasets belonging to the operation. The notation of operation in the related literature is also referred to as *Activity* and *Particle* [1, 3]. Such operations just to name a few are: *Filter*, *Join*, *Sort* etc. Whereas, if we consider the naming conventions of the integration tools already in the market, they use terms such as *Component* (SSIS, Talend), *Step* (Pentaho) or *Operator* (OWB).

As mentioned in the previous section, we have built our work on top of the ETL taxonomy presented in [4]. From the operation complexity point of view, they




(a) A unary ETL atom (b) An ETL molecule

Figure 4.1: Example of an ETL atom and molecule taken from [4]

classify ETL operations into ETL particles and atoms that perform a single transformation; molecules and compound ones that perform more complex transformations.

We simplify this classification into only two main categories, in order to distinguish between atomic operations and compound ones. Similarly to particles, we call *Atomic Operations* those that perform a single transformation. An example of an atomic operation might be *Filter* that is performed completely in one step at the tuple level, and removes those tuples that do not satisfy the filtering condition. On the other hand, non-atomic operations are the *Compound* ones, that perform more than one transformation on the initial dataset, meanwhile having them wrapped up in a single transformation component (as per the implementation details of the integration tools). An example of a compound operation might be *tReplaceList*, which is a component in Talend that performs a value replacement to the original dataset with the values coming from the lookup table. Obviously, it performs both a *Join* operation and an *Attribute value alteration* operation.

From the perspective of operation logic applied to the initial dataset, we use the same categorization used in OWB by classifying ETL operations into two main types:

1. Source and Target operations consisting of the extraction and loading operations

2. Transformation operations that apply changes to the initial dataset(s)

However, our study focuses on the transformation capabilities offered rather than source/target operations.

## 4.2 Data Integration Tools Selection

As discussed in section 3.2., there exist a vast plethora of data integration tools. Out of these plethora, we have selected four; Two pioneer vendors that have been long in the market such as Microsoft SQL Server Integration Services (SSIS)[1] and Oracle Warehouse Builder (OWB)[2] and two newer tools Pentaho Data Integration (Kettle)[3] and Talend Open Studio for Data Integration[4].

The reason behind choosing SSIS and Talend was driven by personal expertize, whereas for Pentaho and OWB the incentive was due to their popularity among developers, as well as the existing work from my colleagues [5] who had already analyzed their respective functionalities.

Each tool has its specifics regarding naming conventions, operations provided, graphical representation etc. However, all of them provide the basic transformation capabilities that are defined in the Gartner report in [25].

## 4.3 Literature Review on ETL Taxonomies

During our literature review, we came across two main ETL operations classification models.

1. First Categorization

   The first one is based on the definitions mentioned earlier and consists of two main types of ETL operations:

[1] Microsoft SQL Server Integration Services
[2] Oracle Warehouse Builder
[3] Pentaho Data Integration
[4] Talend Open Studio for Data Integration
[5] Petar Jovanovic and Vasileios Theodorou

- Source and Target operations consisting of the extraction and loading operations.

  Extraction and loading operations are part of this category. Although they do not form the core part of the data generation problem, for the completeness of our study, we list here the most typically supported formats in the studied ETL tools. Source and target operations are categorized as per the input and output schemata. Below, we have presented only a portion of all possible source and target data stores:

| **Extraction** | **Loading** |
|---|---|
| Relational DBMS | Relational DBMS |
| NoSQL DBMS | NoSQL DBMS |
| Flat File | Flat File |
| XML input | XML output |
| Spreadsheets | Spreadsheets |

Table 4.1: Source and target operations

- Data flow operations

  Operations in this category consist of those that perform actual transformations to the input dataset. Any operation other than the source and target operations fall into the data flow operations category.

2. Second categorization

   Apart from the above classification, in [4] they propose another one which is based on the cardinality of the input and output schemata.

   - Unary: The operation has exactly one input schema and stores the result into one output schema. Alternatively, for the intermediate operations it means that there is exactly one direct preceding operator and one direct succeeding operator.

- N-ary: The operation has many input schemata but produce exactly one output schema. Binary operations are a specific frequent case of the N-ary operations that have two input schemata but one output schema.
- Router: These operations have one input schema but populate more than one output schemata.

## 4.4 Proposed ETL Taxonomy

### 4.4.1 Defining List of ETL Operations

In order to categorize the transformation activities, we have defined our categorization starting from the atomic transformation operations called ETL particles in [4]. As discussed previously, atomic operations are those that offer a single transformation on the input dataset such as: *Projection*, *Sort* etc. In Table 4.2, we have listed the atomic operations considered in our study.

| **ETL Operations** | |
|---|---|
| Aggregation | Join |
| Attribute Addition | Left Outer Join |
| Attribute Alteration | Pivot |
| Attribute Renaming | Projection |
| Cross Join | Right Outer Join |
| Dataset Copy | Router |
| Datatype Conversion | Sampling |
| Difference | Sort |
| Duplicate Removal | Union |
| Replicate Row | Union All |
| Filter | Unpivot |
| Intersect | Value Alteration |

Table 4.2: List of operations considered in the framework

To come up with this list, we have studied the four integration tools presented in the previous section. Most of them provide these transformations as a unique component that performs exactly one atomic transformation at a time. However, there are several other operations that for some tools are not offered as a single component, but instead are embedded in a more complex transformation unit that performs several atomic transformation operations. As an example is the operations called *tMap* in Talend that offers capabilities for several operations i.e., Join, Attribute Addition and Alteration, Router, Filter etc.

In Table 4.3., we present the complete list of ETL operations that we extracted from each tool and the corresponding component name, specific to that tool.

*Note: This table does not contain the complete list of operations from each of the four tools studied. Rather, it contains only those that were significant to our research objective. However, for the completeness of our study, we have added also the source and target operations (extraction and loading) represented by only a part of all possible operations supported.*

| Operation Level | Operation Type | Pentaho PDI | Talend Data Integration | SSIS | Oracle Warehouse Builder |
|---|---|---|---|---|---|
| Source Operation | Extraction | CSV file input<br>Microsoft Excel Input<br>Table input<br>Text file input<br>XML Input | tFileInputDelimited<br>tDBInput<br>tFileInputExcel | ADO.NET/DataReader Source<br>Excel Source<br>Flat File Source<br>OLE DB Source<br>XML Source | Table Operator<br>Flat File Operator<br>Dimension Operator<br>Cube Operator |
| Target Operation | Loading | Text file output<br>Microsoft Excel Output<br>Table output<br>Text file output<br>XML Output | tFileOutputDelimited<br>tDBOutput<br>tFileOutputExcel | Dimension Processing<br>Excel Destination<br>Flat File Destination<br>OLE DB Destination<br>SQL Server Destination | Table Operator<br>Flat File Operator<br>Dimension Operator<br>Cube Operator |
| Value | Single Value Alteration | If field value is null<br>Null if<br>Modified Java Script Value<br>SQL Execute | tMap<br>tReplace | Derived Column | Constant Operator<br>Expression Operator<br>Match-Merge Operator<br>Mapping Input/Output parameter |
| Attribute | Attribute Value Alteration | Add constant<br>Formula<br>Number ranges<br>Add sequence<br>Calculator<br>Add a checksum | tMap<br>tConvertType<br>tReplaceList | Character Map<br>Derived Column<br>Copy Column<br>Data Conversion | Constant Operator<br>Expression Operator<br>Data Generator<br>Transformation<br>Mapping Sequence |
| Tuple | Duplicate Row | Clone Row | tRowGenerator | | |
| Tuple | Filter | Filter Rows<br>Data Validator | tFilterRow<br>tMap<br>tSchemaComplianceCheck | Conditional Split | Filter |
| Tuple | Join | Merge Join<br>Stream Lookup<br>Database lookup<br>Merge Rows<br>Multiway Merge Join<br>Fuzzy Match | tJoin<br>tFuzzyMatch | Merge<br>Join<br>Fuzzy Lookup | Joiner<br>Key Lookup Operator |
| Tuple | Router | Switch/Case | tMap | Conditional Split | Splitter |
| Tuple | Set Operation-Intersect | Merge Rows (diff) | tMap | Merge Join | Set Operation |
| Tuple | Set Operation-Difference | Merge Rows (diff) | tMap | | Set Operation |
| Tuple | Set Operation-Union | Sorted Merge<br>Append streams | tUnite | Merge<br>Union All | Set Operation |

| Operation Level | Operation Type | Pentaho PDI | Talend Data Integration | SSIS | Oracle Warehouse Builder |
|---|---|---|---|---|---|
| Dataset | Duplicate Removal | Unique Rows<br>Unique Rows (HashSet) | tUniqRow | Fuzzy Grouping | Deduplicator |
| Dataset | Sort | Sort Rows | tSortRow | Sort | Sorter |
| Dataset | Sampling | Reservoir Sampling<br>Sample Rows | tSampleRow | Percentage Sampling<br>Row Sampling | |
| Dataset | Aggregation | Group by<br>Memory Group by | tAggregateRow<br>tAggregateSortedRow | Aggregate | Aggregator |
| Dataset | Dataset Copy | | tReplicate | Multicast | |
| Schema | Attribute Addition | Set field<br>value<br>Set field value to a constant<br>String operations<br>Strings cut<br>Replace in string<br>Formula<br>Split Fields<br>Concat Fields<br>Add value fields changing sequence<br>Sample rows | tMap<br>tExtractRegexFields<br>tAddCRCRow | Derived Column<br>Character Map<br>Row Count<br>Audit Transformation | Constant Operator<br>Expression Operator<br>Data Generator<br>Mapping Input/Output<br>parameter |
| Schema | Datatype Conversion | Select<br>Values | tConvertType | Data<br>Conversion | Anydata<br>Cast Operator |
| Schema | Attribute Renaming | Select<br>Values | tMap | Derived<br>Column | |
| Schema | Projection | Select Values | tFilterColumns | | |
| Relation | Pivoting | Row Denormalizer | tDenormalize<br>tDenormalizeSortedRow | Pivot | Unpivot |
| Relation | Unpivoting | Row Normalizer<br>Split field to rows | tNormalize<br>tSplitRow | Unpivot | Pivot |

Table 4.3: Transformations provided by four main ETL tools

### 4.4.2 Proposed ETL Taxonomy

After defining the complete list of operations of interest in our study, we categorize them based on the taxonomy provided in the literature review section. Therefore, we mapp each operation to the respective categories of: Unary, N-ary (with Binary being a special case) and Router operations. We consider Binary operations separately as a special case of N-ary operations, since there are operations such as Difference, Intersect that can only have two input relations. Moreover, we focus on binary operations since N-ary operations can be expressed as a composition of binary ones. This categorization is important for processing the flow and tracking schema transformations and cardinality changes.

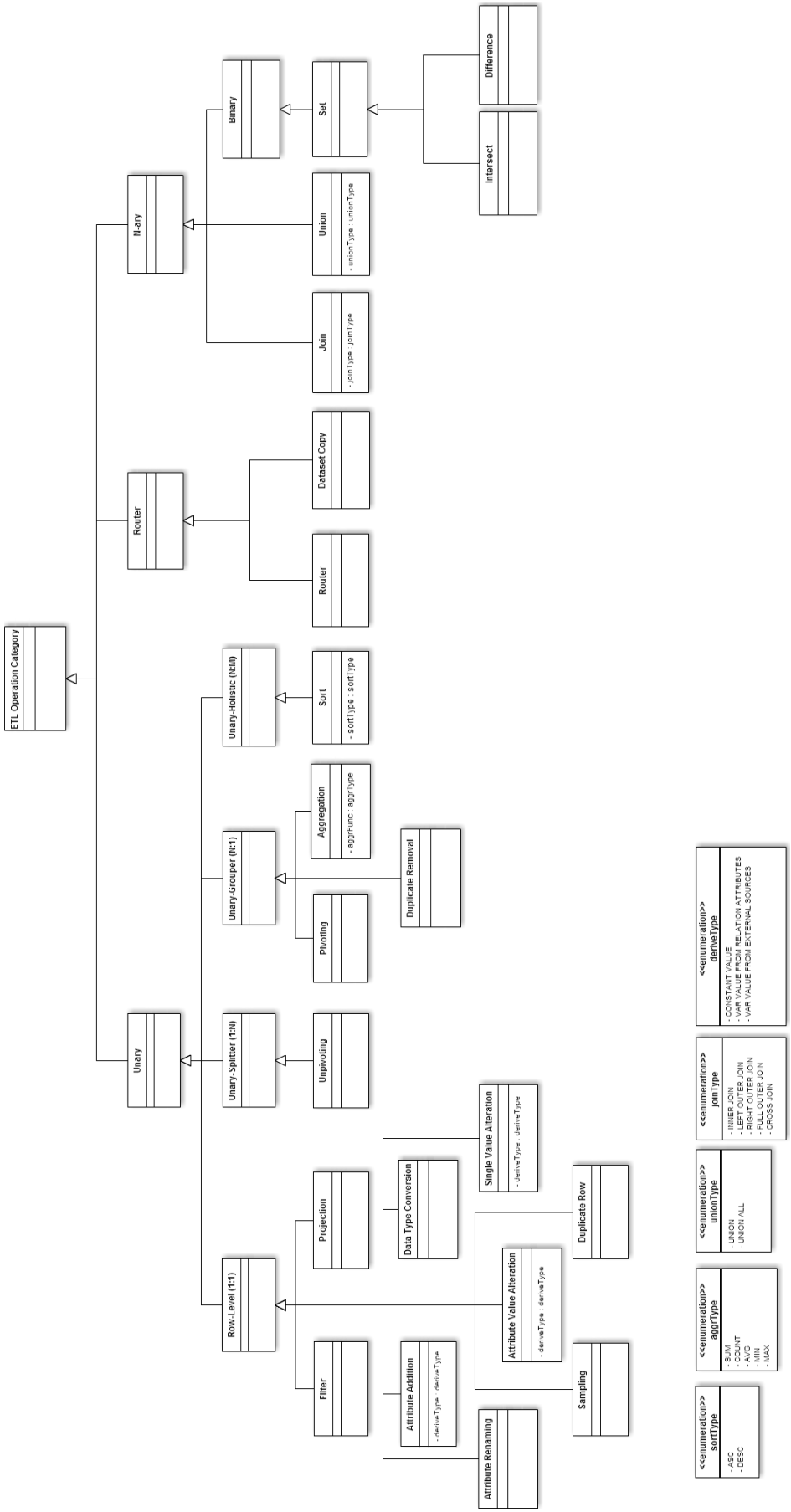



Figure 4.2: Adapting taxonomy from [4] to our scope

In addition, we propose another taxonomy which is specific and particularly valuable for the context of ETL data generation. It is based on the properties of the input relation(s) that are accessed by each operation in order to perform the respective changes. With the term *relation properties* we refer to the level of detail consumed such as: tuple level, attribute level or schema level etc. Next paragraph is devoted to the introduction of such idea, along with a graphical illustration.

In Figure 4.3, we illustrate our model of classifying the relation levels of detail. However, needed to stress is that the scope of this thesis is not restricted only to the relational database model, because it is assumed the input and output schemata can be either relational or not. Instead, relation refers to any input or output schema regardless of the underlying implementation constraints. Similarly, we do not limit the scope of this work only to relational algebra operations.

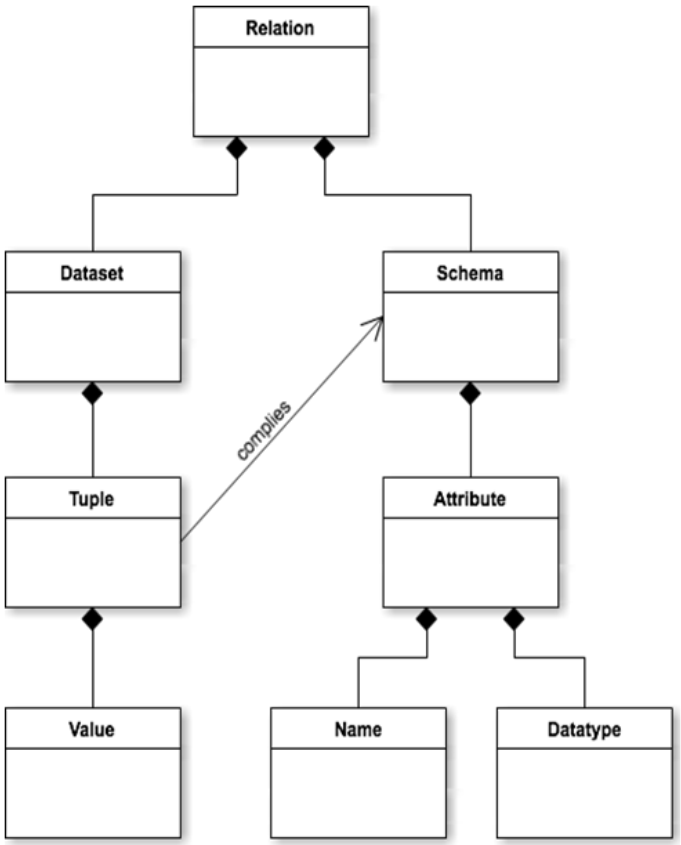



Figure 4.3: Relation access level taxonomy

The importance of this type of categorization is that it gives valuable information for the data generation process. It provides those properties (level of detail as

referred to previously) of the input relation we are looking for when generating the source data. Also, it gives information on how the schema is changing from one operation to the other.

The idea behind this classification model is to map each level to the corresponding ETL operations that access it, in order to perform the transformation. Alternatively, it consists of the knowledge prerequisites that the operation needs to have regarding the dataset as a whole, schema or tuple only etc. For instance: the *Filter* operation partitions the dataset based on a filtering condition. As a result, we classify this operation at the tuple level since the transformations are performed tuple by tuple. Also during the data generation, based on the filtering condition, we can generate one tuple at a time.

At the top level stands the relation, which is composed of the dataset holding all the values of the relation and the corresponding schema. Operations at the relation level are those that perform value transformation as well as schema modification. It can be seen as an operation that completely blocks the entire relation while being executed.

The dataset itself comprises of tuples (also referred to as entry or record), where each holds values for only one entity of the dataset. Operations at the dataset level consume the entire set of tuples in order to perform the transformations. One typical operation is *Sort* which is called holistic in [4] exactly for specifying this tuple blocking property.

Operations at the tuple level are performed horizontally one record at a time. Hence, they do not require the entire relation to be blocked, but rather only the tuple they are working on.

Whereas, the schema is composed of the attributes, which in turn have two other properties: attribute name and attribute datatype. Operations at the schema level obviously change the schema of the initial relation either by removing or adding new attributes. While, those operations that apply changes only to the values of a particular attribute of the dataset without actually changing the schema belong to the attribute level.

To summarize what was presented above, the operations can perform transfor-

mations in one of the corresponding layers:

- Relation level operation

  Operations that require to make changes to the entire relation, both value and schema-wise correspond to the category of Relation level operations. Some typical examples are: *Row Denormalizer (Pivot)* and *Row Normalizer (Unpivot)*. Both these operations change the schema by adding or removing attributes as well as the content of the original dataset.

- Dataset level operation

  Operations at the dataset level are those that access the input relation as a whole, but change only the values, do not modify the schema. Typical examples are blocking operations such as: *Sort* and *Duplicate removal* that need to access the entire dataset prior to applying the corresponding transformations.

- Schema level operation

  Operations at the schema level operate on the input schema and modify it by removing or adding new attributes. Some examples of operations corresponding to this category are *Projection* and *Attribute Addition*.

- Tuple level operation

  Operations at the tuple level access the input schema tuple by tuple and apply the transformations one tuple at a time. Typical entry level operation is *Filter* that checks each row whether it satisfies the condition or not. In case the condition is not met, these rows are filtered out while the rest is passed to the output schema.

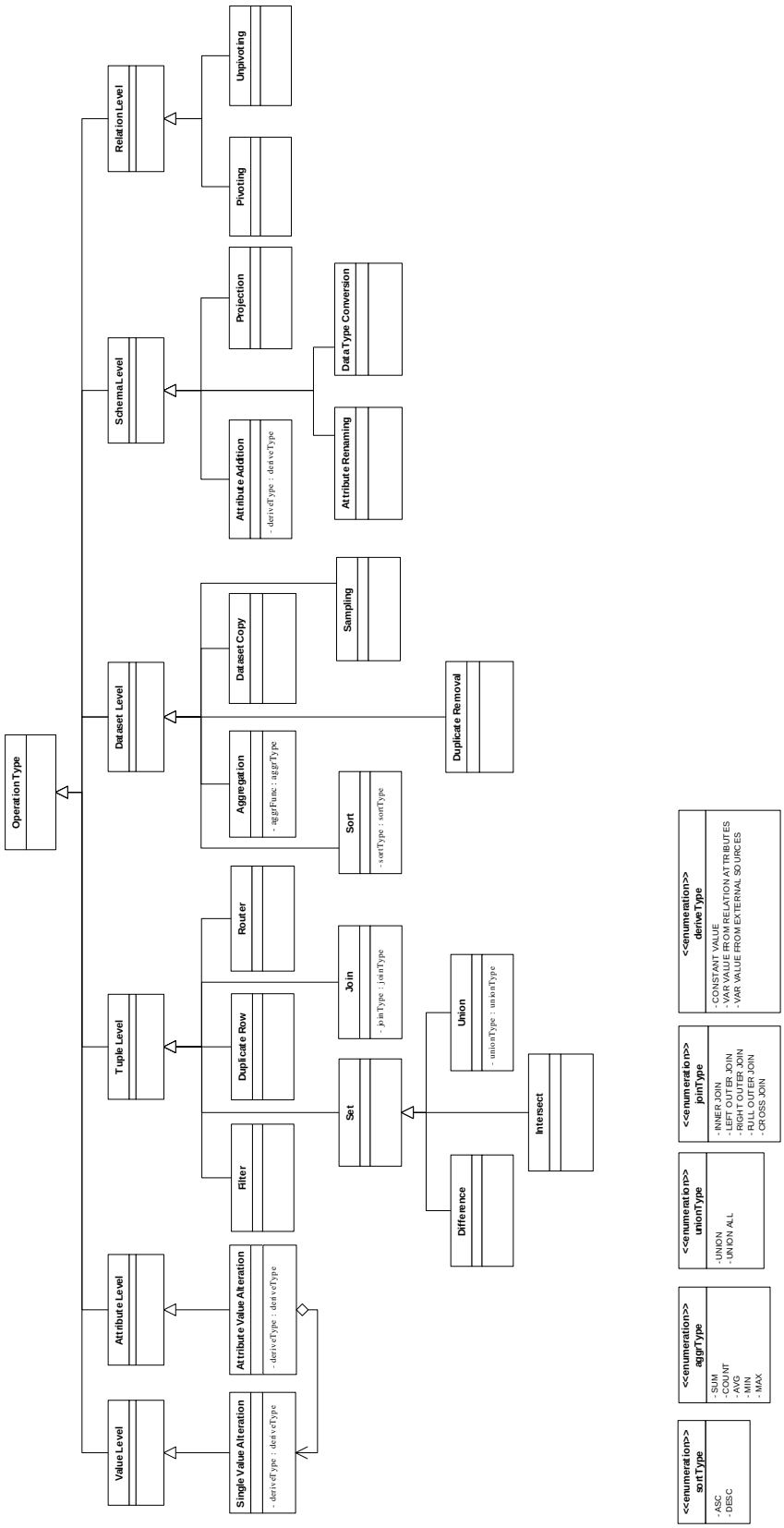
- Attribute level operation

  Operations at the attribute level access the input dataset column-wise, at specific given attributes. *Attribute value alteration* is an ETL operation that falls under this category since the alteration transformation is applied to the entire column (attribute) at once.

- Value level operation

  Operations at the value level aim at replacing single values from the relation with a new one. The new value can either be a constant, or a functional derivation from the relation's data or other external data sources. These operations do not deal with an entire row or column, just with single values, and consequently they are classified under value level operations.

In continuation we graphically illustrate our own ETL taxonomy for the context of our data generation problem. The naming convention for the operations is a generalization of the names found in data integration tools. In some other cases we have adopted the naming convention of one of the data integration tools.


Operation Type
Value Level
Attribute Level
Tuple Level
Dataset Level
Schema Level
Relation Level
Single Value Alteration
- deriveType : deriveType
Attribute Value Alteration
- deriveType : deriveType
Filter
Duplicate Row
Router
Set
Join
- joinType : joinType
Difference
Union
- unionType : unionType
Intersect
Aggregation
- aggrFunc : aggrType
Sort
- sortType : sortType
Dataset Copy
Sampling
Duplicate Removal
Attribute Addition
- deriveType : deriveType
Projection
Attribute Renaming
Data Type Conversion
Pivoting
Unpivoting
<<enumeration>>
sortType
- ASC
- DESC
<<enumeration>>
aggrType
- SUM
- COUNT
- AVG
- MIN
- MAX
<<enumeration>>
unionType
- UNION
- UNION ALL
<<enumeration>>
joinType
- INNER JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN
- CROSS JOIN
<<enumeration>>
deriveType
- CONSTANT VALUE
- VAR VALUE FROM RELATION ATTRIBUTES
- VAR VALUE FROM EXTERNAL SOURCES

Each of the operations deriving directly from the first layer is called *Operation Type* and they comprise the second level of our taxonomy. Each operation type itself can have several subtypes which in turn are enumerated at the bottom left side of the diagram. For instance, the *Attribute Alteration* itself is an operation type. But, it can alter the given attribute with a constant or with a variable value derived from a given function which might be calculated from internal or external resources. As a result, we distinguish three different cases or subtypes for the attribute alteration operation.

In the next section we describe each operation level separately.

### 4.4.3 Value Level Operations

Value level operations perform their actions on specific values of the dataset. They do not access the entire tuple neither the entire column - only specific values. We call the operation under this category *Single Value Alteration* and it replaces those values satisfying a given condition with a new one.

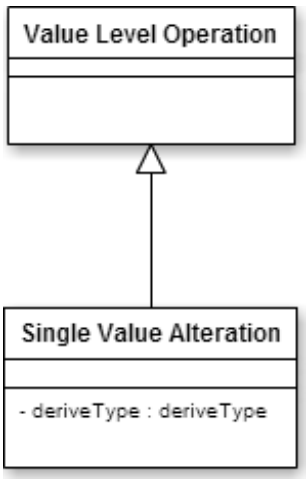



Figure 4.5: Value-level operations

### 4.4.4 Attribute Level Operations

Operations at the attribute level access the input dataset column-wise, at specific given attributes. Attribute value alteration is an ETL operation that falls under this category since the alteration transformation is applied to the entire column.

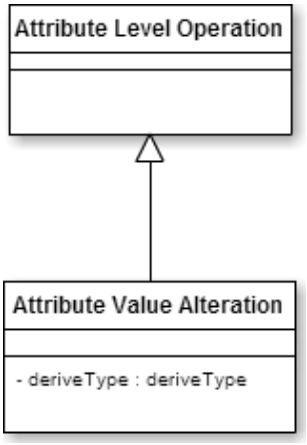



Figure 4.6: Attribute-level operations

### 4.4.5 Tuple Level Operations

These operations access the input relation tuple-wise or entry-wise. They perform actions on the entire tuple at once. They can be unary, binary or routing operations. *Filter* and *Router* are examples of unary and router types of operation that partition horizontally the input relation by filtering out those tuples that do not satisfy some given conditions. Other binary operations that are performed tuple-wise are join operations: *Inner Join, Outer Join* and set operations: *Union, Intersect and Difference.*

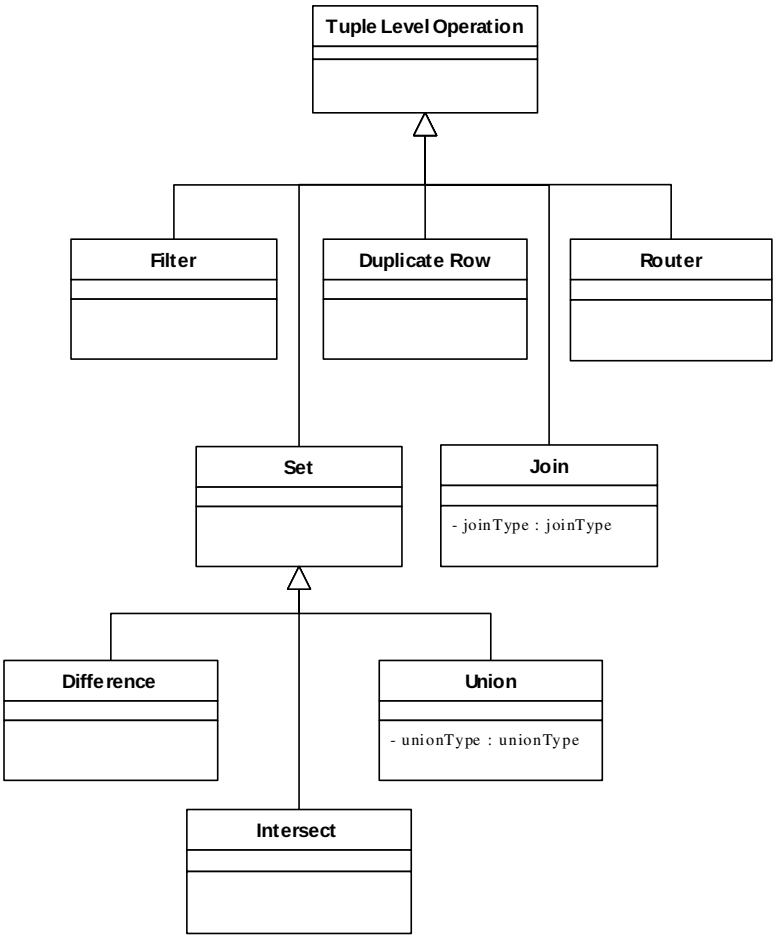



Figure 4.7: Tuple-level operations

### 4.4.6 Dataset Level Operations

Operations at the dataset level are those that access the input relation as a whole. Typical examples are blocking operations such as: *Sort* and *Aggregation* that need to access the entire dataset prior to applying the corresponding transformations.

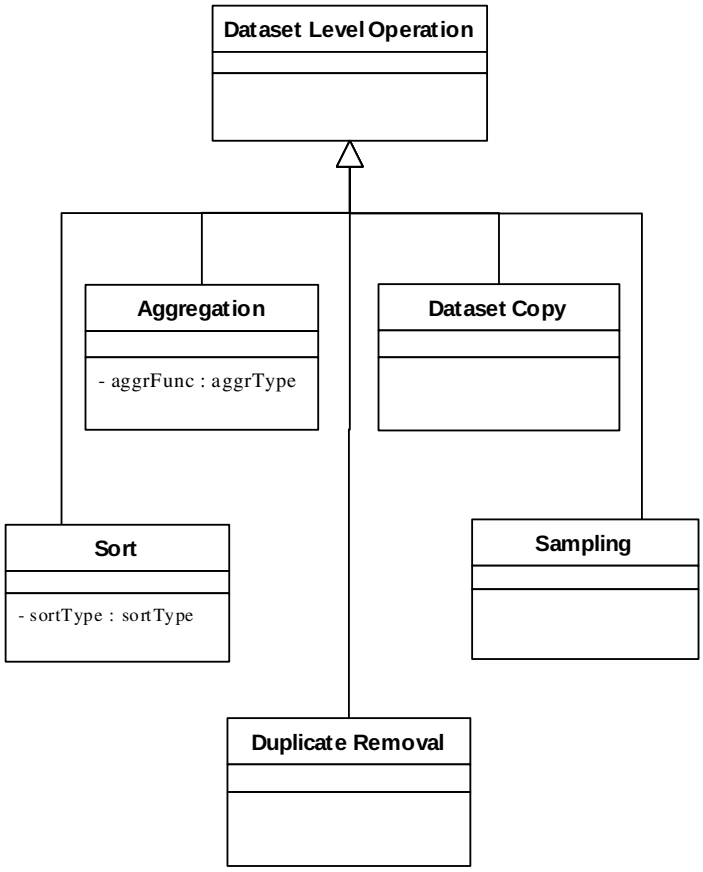



Figure 4.8: Dataset-level operations

### 4.4.7 Schema Level Operations

Operations at the schema level operate on the input schema and modify it by removing or adding new attributes. Atomic operations corresponding to this category are *Projection* and *Attribute Addition*, as well as *Datatype Conversion* and *Attribute Renaming*.

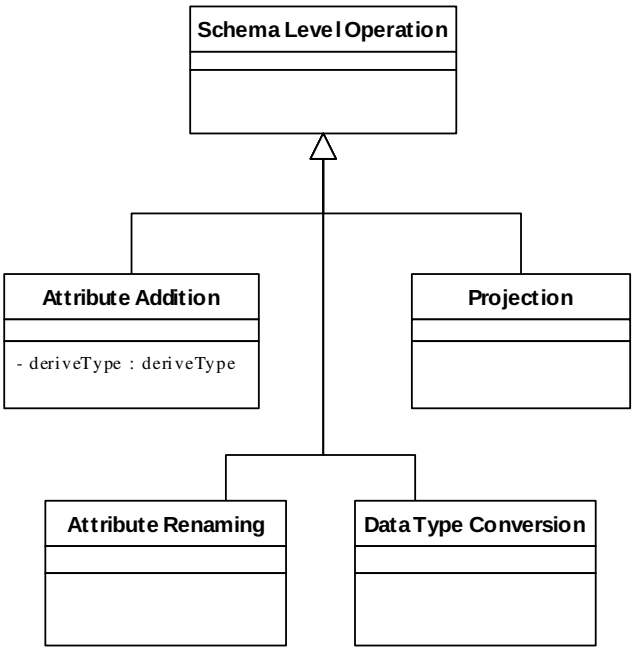



Figure 4.9: Schema-level operations

### 4.4.8 Relation Level Operations

These operations make changes to the entire relation, both dataset and schema-wise. This is the category of relation level operations and some typical examples are: *Pivoting* and *Unpivoting*. Both these operations change the schema by adding or removing attributes as well as the content of the original dataset.

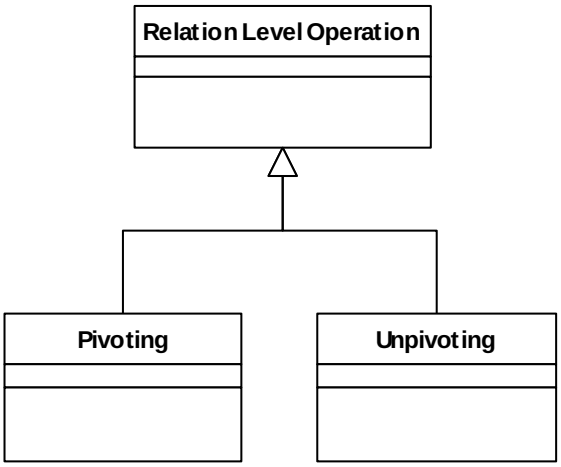



Figure 4.10: Relation-level operations

## 4.5 ETL Operation Semantics Definition

As the second contribution of our work, after the categorization of ETL activities, we analyze them from the schema transformation point of view. We describe the semantics of ETL activities from two aspects: schema transformations (mappings between input and output schemata) and tuple transformation. We have based our analysis on the previous work from [4], [52].

In [4], the authors model the ETL activity as a pentad of the form (I,m(),P(X),r,O) where:

- I is a finite set of (input) schemata,
- m is a merger,
- P(X) is a materialization of a template predicate over the schema X, which we call functionality schema of the atom,
- X is a subset of the union of attributes of the schemata of I,
- r is a router,
- O is a finite set of (output) schemata.

Whereas, in [52] they extend the same notation principles by adding more parameters and more expressiveness to the ETL activity definition (In,Out,T,F,Po,Cc,Gen), where:

- In is input relations,
- Out is output relations,
- F is the operation of act,
- T is the classification,
- Po contains projected-out attributes,
- Cc is the set of additional concerned attributes taking part in F,
- Gen is the set of generated attributes.

Inspired by the above ETL semantic definitions, we model ETL transformation operation as ($\mathbb{I}$,$\mathbb{O}$,$\mathbb{X}$,$\mathbb{S}$,A), where:

- $\mathbb{I}$ is a final set of Input relations,
- $\mathbb{O}$ is a final set of Output relations,
- $\mathbb{X}$ is a vector of attributes used in the operation semantics,
- $\mathbb{S}$ is the set of semantics applied over the input schema, which might be a predicate or a function,
- A is a vector of attributes from the output relation that were added or altered during the operation.

The complete list of notations used is presented in Table 4.4.

We define the operation semantics using the above symbolic representation. We denote the transformation semantics as a quintuple of *($\mathbb{I}$,$\mathbb{O}$,$\mathbb{X}$,$\mathbb{S}$,A)*. This notation defines the transformations of the schemata of the input *($\mathbb{I}$)*, into the result schemata

| Notation | Description |
|---|---|
| $\mathbb{I}=\{I_1, \ldots, I_n\}$ | Set of input relations. $I_i$ is the $i^{th}$ input relation. |
| $I=I_1$ | Unary operator. |
| $\mathbb{I}=\{I_1, I_2\}$ | Binary operator. |
| $\mathbb{I}=\{I_1, \ldots, I_n\}$ | N-ary operator. |
| $SI_i$ | Schema of the $i^{th}$ input relation. $SI=\{a_1, \ldots, a_n\}$ |
| $\mathbb{O}=\{O_1, \ldots, O_n\}$ | Set of output relations. $O_j$ is the $j^{th}$ output relation. |
| $O=O_1$ | One output relation. |
| $\mathbb{O}=\{O_1,\ldots, O_n\}$ | Multiple output relation. |
| $SO_j$ | Schema of the $j^{th}$ output relation. $SO=\{b_1, \ldots, b_n\}$ |
| $\mathbb{X}=\{X_1, \ldots,X_n\}$ | Set of attributes consumed during the operation. $X_i \subset SI_i$ |
| $t_i$ | Tuple of the input dataset. |
| $t_o$ | Tuple of the output dataset. |
| $t_i[X_j]$ | Values of the set X of attributes from the $i^{th}$ tuple of the $j^{th}$ input relation. |
| $\mathbb{S}$ | Set of semantics applied on the input schema. |
| $S(t_i[X_j])$ | Semantics (i.e., predicate, function) over the values of subset X of the $j^{th}$ input schema from the $i^{th}$ tuple. |
| $\mathbb{S}=[S_1(X_1), \ldots, S_n(X_n)]$ | Set of semantics over n input relations. |
| A | Set of attributes from the output relation that were added or altered. $A \subseteq SO$ |
| $\lvert I_i \rvert$ | Cardinality of the $i^{th}$ input dataset. |
| $\lvert O_i \rvert$ | Cardinality of the $i^{th}$ output dataset. |
| $\lvert SI_i \rvert_a$ | Arity of the $i^{th}$ input schema. |
| C | Constant value. |
| $F(\mathbb{I},\mathbb{O},\mathbb{X},\mathbb{S},A)$ | Represents the type of operation applied over the quintuple $(\mathbb{I},\mathbb{O},\mathbb{X},\mathbb{S},A)$. |

Table 4.4: Table of operation semantics notations

of *($\mathbb{O}$)*, by applying semantics *($\mathbb{S}$)*, along with the mappings of input attributes *($\mathbb{X}$)* to the output's attributes *(A)*.

These semantics express schema transformations as well as changes on the tuple level. In addition, we express also cardinality requirements of the output schema with respect to the cardinality of the input schema using the same semantics. Semantics are expressed via two expression formulas comprising of two sides each. The first expression shows the semantics of schema and cardinality requirements that need to hold in order for the operation to take place. In the left-hand side of the expression are shown the parameters of the quintuple and the exact type of operation applied on them. The right side shows the schema and cardinality requirements for that particular operation, whereas the second expression shows the actual changes undergone by the operation itself at the tuple and attribute level.

Alternatively, starting from the left-hand side of the expression, it can be read as: What is given and which operation is applied, and at the right part of the operation we show the resulting output schema and which are the transformations occurred with respect to the initial state.

To clarify, let us look at an example of ETL operation at the value level, which alters the values based on a condition[6]. In this case, operation level is *Value Level* and operation type is *Single Value Alteration*. As it was introduced in our ETL taxonomy, we distinguish three separate cases:

1. The new value replacing the old one is a constant
2. The new value replacing the old one is derived from some of the attributes of the same relation
3. The new value replacing the old one is derived from an external function

Thus, we have three different operation subtypes:

1. Constant value alteration

[6]In the Appendix section we introduce a more complete list of semantics for the most representative ETL operations from each operation type from our proposed taxonomy.

2. Relation dependent value alteration

3. External data dependent value alteration

Let us consider the second case, of deriving the new value from the relation itself. The operation semantics we identify for this case are:

$\forall$(I,O,X,$\mathbb{S}$,A) (F(I,O,X,$\mathbb{S}$,A) $\rightarrow$ (SO=SI $\wedge$ |O|=|I|))

$\forall t_i \in$I ($S_1(t_i[X]) \rightarrow \exists t_o \in$O ($t_o$[SO \ A]=$t_i$[SI \ A] $\wedge$ $t_o$(A)=$S_2(t_i[X])$)))

- F – The operation applied to the quintuple *(I,O,X,$\mathbb{S}$,A)*, which corresponds to the operation type/subtype. In this particular example the operation type is *Value alteration* and subtype *Relation dependent value alteration*,

- I – Unary operator because we have only one input schema,

- O – One resulting output schema,

- X – Subset of the union of attributes from the input schema, considered (used) during the functional derivation of the new value of attribute ($A$) and for the condition verification,

- $\mathbb{S}$ – Set of semantics: $S_1(t_i[X])$ - the condition predicate and $S_2(t_i[X])$ - function applied over the attributes of the input schema ($t_i[X]$), in order to replace the old values of attribute ($A$),

- A – Attribute corresponding to the value that is subject to the modification,

- SO=SI – No schema modification. Schema of both input and output relations are the same.

We interpret these semantics as follows:

1. For all quintuples *(I,O,X,$\mathbb{S}$,A)* over which we apply the operation of type F, must hold that the output schema is equal to the input schema and that the cardinality of the output dataset is equal to the cardinality of the input dataset.

2. For every tuple of the input relation ($I$) that satisfies the condition predicate ($S_1(t_i[X])$), there exists a tuple from the output relation $O$ such that they are equal for each attribute other than the one whose value is being changed ($t_o[SO \setminus A]=t_i[SI \setminus A]$); Whereas, the attribute ($A$) of the new value is derived by the formula applied over the attributes ($X$) of input schema ($I$), by applying the function denoted as ($S_2(t_i[X])$).

## 4.6 Solution Limitations

In this study we consider only ETL flows that are correct, in the sense they are satisfiable for all constraints and all branches are visited, given the right input.

### 4.6.1 Atomic Operations

In our classification model we consider only what we above refer to as *Atomic Operations*. The main reason of not including *Compound Operations* in the model is due to the fact that in general they are platform dependent and thus found in a specific integration tool. Whereas, our model tends to be generic and applied to most operations found in the ETL flow, independently of the underlying implementation specifications.

In addition, we aim at analyzing each transformation step individually in order to understand the way it operates, in terms of requirements specification, transformation semantics and expected outcome. By analyzing each operator separately, we build the environment for analyzing the compound operations from these atomic ones. The reason is that it is possible to represent compound operation as a combination of atomic ones.

### 4.6.2 Logical Predicates

In order to resolve the semantics of the operations and represent them by the first order logic symbolism, we use logical predicates for representing the transformation rules. Such a predicate can be for example the filtering condition "*Age greater than 25*". This a simple form of a predicate applied over the values of attribute *Age* from

the tuples of the input schema, which can be presented by *S(t[Age]*. So in general, simple predicates are those that apply a singular function over the input dataset, which might be a simple mathematical formula or some basic string processing transformations.

But, predicates can get more complicated when multiple transformations are applied over the initial dataset. This may include for instance a combination of several complicated mathematical functions. We call them complex predicates and represent as $S=[S_1(X_1), \ldots, S_n(X_n)]$. This notation means that we are dealing with a complex predicate which is composed of several simple predicates $S_1$, $S_2$, ... and so on, connected by logical operators: *AND, OR, XOR, NOT*.

However, we limit the scope of our study to only conjunctive predicates that are composed of several simple predicates connected with *AND* operands.

### 4.6.3 Custom-defined Transformations

Custom-defined transformations are excluded from the scope of this thesis. In general, their transformation characteristics are defined by the user based on their specific requirements. An example would be *Database procedure calls* which makes a call to external resources (e.g., function, store procedure etc.) in order to perform a transformation. They cover a broad range of transformations encapsulated within a single transformation step, thus resulting into complex operation that are difficult to evaluate from our side. Since they are not generic but rather user and tool dependent and generally not atomic, they are excluded from the scope of this thesis.

### 4.6.4 Tool-specific Operations

As already mentioned, integration tools differ from each other because of the implementation details as well as modeling capabilities. Each of the tools provides the basic generic transformations, but in addition, they also provide other operations particular to the tool itself. However, for the purpose of our study we have not considered such operations specific to one particular tool since we aim at . Similarly

we exclude from our scope also operations that do not perform any transformation, but rather validate the circumstances for the operation to take place, such as: *Column/table existence check*. The reason is that we consider the flow to be valid and not fail due to a missing input file, attribute etc.

## 4.7 Extensibility

Although we have limited the scope of our study to only atomic operations whose semantics consist of simple predicates, we need to stress that still our model offers extensibility capabilities in both ways of:

1. Extending the taxonomy with other operations

First of all, compound operations can be added to our **ETL taxonomy** *presented in section 4.6* and represented as a composition of atomic operations already defined in the model. The main reason of not including such complex operations in the model is due to the fact that in general they are platform dependent and thus found in a specific integration tool. Whereas, our model tends to be generic and applied to most operations found in the ETL flow independently of the underlying implementation specifications.

The same goes for scripting functionalities of ETL components. We do not consider them during our analysis of the flow since they are not generic. Instead, they are unpredictable and cover a broad range of transformations encapsulated within.

2. Extending the semantics to expressing also non-atomic and more complicated operations

Secondly, the conditions applied over the datasets in the flow can range from a simple predicate such as "*Attribute Age should be greater than 20*" to more complex ones. The latter can consist of complicated mathematical functions applied over the data and/or conjunctions or disjunction of several predicates altogether. In order to simplify our study, we limited the work to only simple predicates consisting of simple mathematical functions such as: $<$, $\leq$, $\geq$, $>$, $=$, $\neq$ and connected only with the AND operand which simplifies reasoning over such logical predicates. However,

our model fully offers the capability to support also such complex predicates.

# 5 ARCHITECTURE DESIGN

In this chapter we present an overview of the proposed architecture for the data generation framework.

## 5.1 ETL Data Generation Architecture

### 5.1.1 Input

The main input of our framework is the ETL process. As it was already discussed thoroughly in Chapter 2, we base our prototype on the logical (platform-independent) modeling presented in [1]. Based on their formalization, ETL processes are modeled as directed acyclic graphs (DAG). Along with the data flow, we assume that an ETL process carries different parameters that can be used to lead the process of data generation. Such parameters can be found on different levels of the process, i.e., (1) attribute level (e.g., datatype and distribution function), (2) operation level (e.g., operation selectivity factor). Besides ETL process' parameters, the input to our framework can be additionally customized to provide more configuration parameters for data generation process (e.g., load size). Notice that the framework is extensible to larger set of configuration parameters, both at the input and within an ETL process.

We design the architecture of the framework as a layered architecture which is depicted in Figure 5.1.

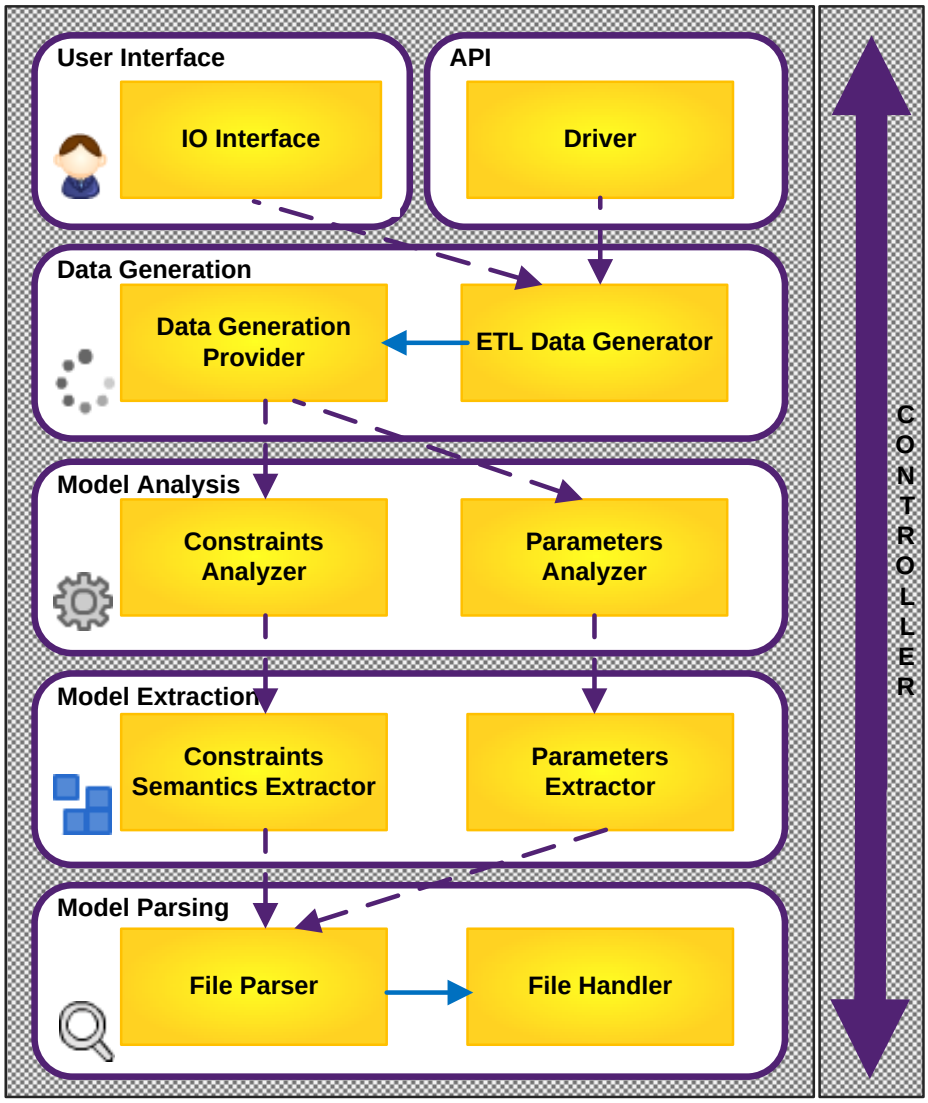
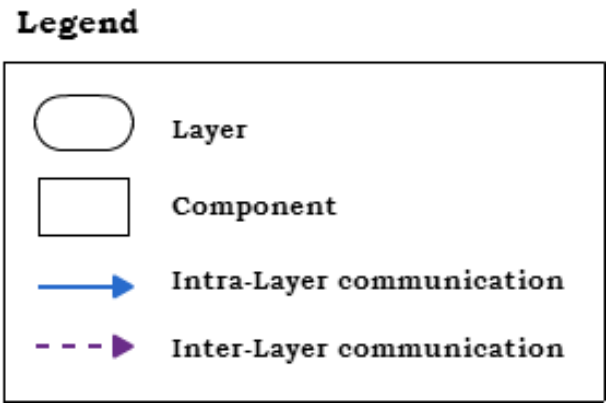



Figure 5.1: ETL data generator framework architecture



We devote a separate section for each of the layers, in order to explain their components and functionalities in more details.

### 5.1.2 Model Parsing

The bottom layer of the architecture is responsible for parsing the model of the ETL process from the given logical representation of the flow (e.g., XML), and importing a DAG representation for the process into the framework. In general, the Model Parsing layer can be extended with external parsing plugins for handling different logical representations of an ETL process (e.g., [1]).

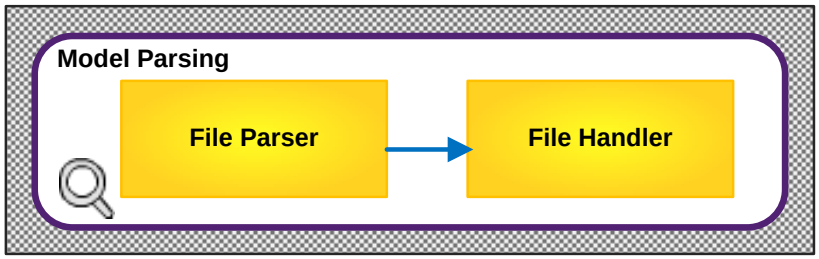



Figure 5.2: Model parsing layer

Model Parsing layer is implemented by two components:

1. File Handler

   This component deals with the task of importing the logical representation model of the flow (e.g., XML) uploaded by the user into our framework. The imported model is later processed by the top layers for extracting parameters and flow semantics.

2. File Parser

   This component realizes the parsing of logical representation of ETL (e.g., XML), and transforms it into a DAG model where nodes represent ETL activities (data stores and operations) while edges represent the relationship between them. This component communicates with the *File Handler*, in order to provide us with the parsing of the DAG model. Once parsed the model,

it can be used to extract useful information relevant for the data generation algorithm.

### 5.1.3 Model Extraction

On top of *Model Parsing* there is the *Model Extraction* layer which directly communicates with the bottom one, in order to extract the relevant information needed to process the ETL flow. The relevant information extracted and previously parsed includes: information about input schemata, operations present in the flow, ordering of operations in the flow. Moreover, it provides relevant information about each operation and corresponding properties such as: operation semantics, schema changes and other parameters for data generation. This information retrieved from the parser is later used inside our generation algorithm. Two components implement this layer:

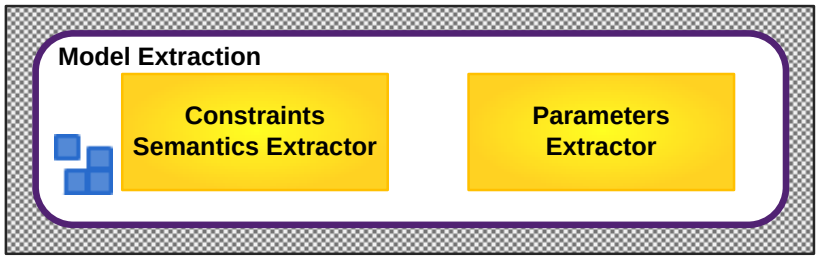



Figure 5.3: Model extraction layer

1. Constraint Semantic Extractor This component is responsible for retrieving operations constraint semantics. The extracted information feeds other components that handle the analysis task.

2. Parameter Extractor Besides semantics extraction, we also extract other model properties. As we mentioned previously, these properties can be found on different levels of the process, i.e., (1) attribute level (e.g. datatype, distribution), (2) operation level (e.g. operation selectivity factor). Specifically, the *Parameter Extractor* component extracts these parameters.

It is to be noticed that this layer can be extended to support a larger set of configuration parameters.

### 5.1.4 Model Analysis

*Model Analysis* layer communicates with the previous one in order to carry further analysis over the previously extracted information.

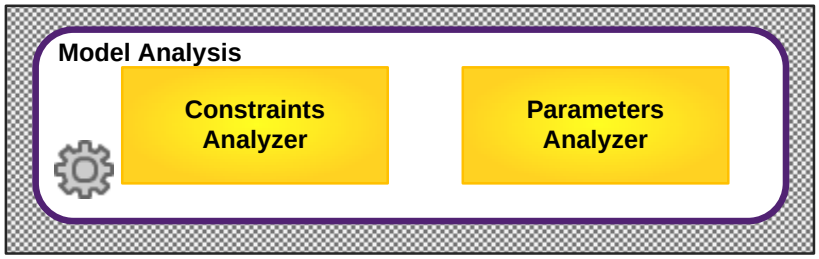



Figure 5.4: Model analyzer layer

This layer is implemented by two components:

1. Constraints Analyzer

   This component analyzes the semantics of operations in order to understand the constraints that are applied, which attributes are being consumed or changed, and how the schema is changed. It directly communicates with the *Constraint Semantic Extractor* component from the *Model Extraction* layer.

2. Parameters Analyzer

   The *Model Parameters Analyzer* component is responsible for analyzing the other configuration parameters extracted by the *Model Parameters Extractor* component.

### 5.1.5 Data Generation

The next layer is *Data Generation*, which controls the data generation process by following the semantics of the constraints and parameters extracted and analyzed by the previous layers.

There are two components implementing this layer:

1. Data Generation Provider

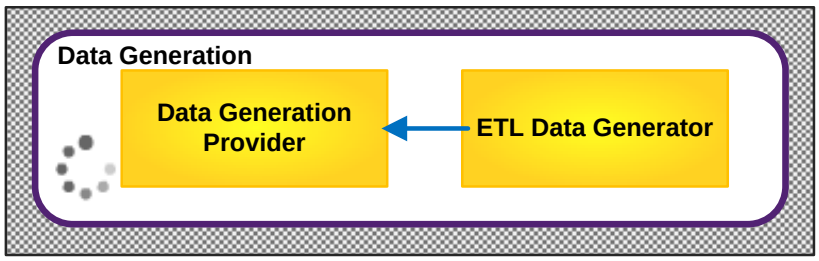



Figure 5.5: Data generation layer

This component provides the basic data generation functionality, and is responsible for generating data values based on the previously extracted and analyzed information about the process configuration properties (e.g. attribute distribution, datatype).

2. ETL Data Generator

   The main engine of the data generation is the *ETL Data Generator* component. It communicates with the *Data Generation Provider* and is responsible for generating the final dataset. While *Data Generator Provider* provides the data generation following properties at the attribute level, the *ETL Data Generator* component is responsible for generating the final dataset following also the properties at the operation level (e..g., selectivity) and load size. Moreover, another important functionality of this component is to construct the final output of the generated data in a compatible format with the input datasets schemata.

### 5.1.6 Presentation Layer

Finally, on the top of the architecture stack we provide the *Presentation*, which consist of the *User Interface* and the *API*.

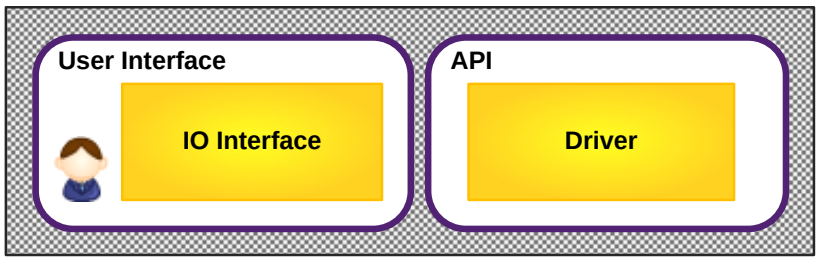



Figure 5.6: Presentation layer

1. User Interface *User Interface* manages the communication between the framework and the end-users of the framework. It guides the users in the process of importing ETL process models and selecting desired parameters for the data generation.

2. API Additionally, the framework can expose its functionality to an external system (e.g., design and execution tools, optimizers, etc.) through the *API* component.

### 5.1.7 Controller

The controller coordinates the communication among different layers in the way similar to computer communication bus. The *Controller* contains the business logic of the application and is responsible for the orderly execution of the algorithm by orchestrating the various components. In this respect, it coordinates the supply of data between layers and takes care of error handling operations.

### 5.1.8 Output

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

The functionality of the main components of our framework (i.e., extraction, analysis and generation) are discussed in more details in the following section related to the data generation algorithm.

## 5.2 Data Generation Algorithm

In this section we introduce the data generation algorithm and also the process flow followed until the final generated output is produced.

### 5.2.1 Process Flow

The data generation process flow is depicted in Figure 5.7.

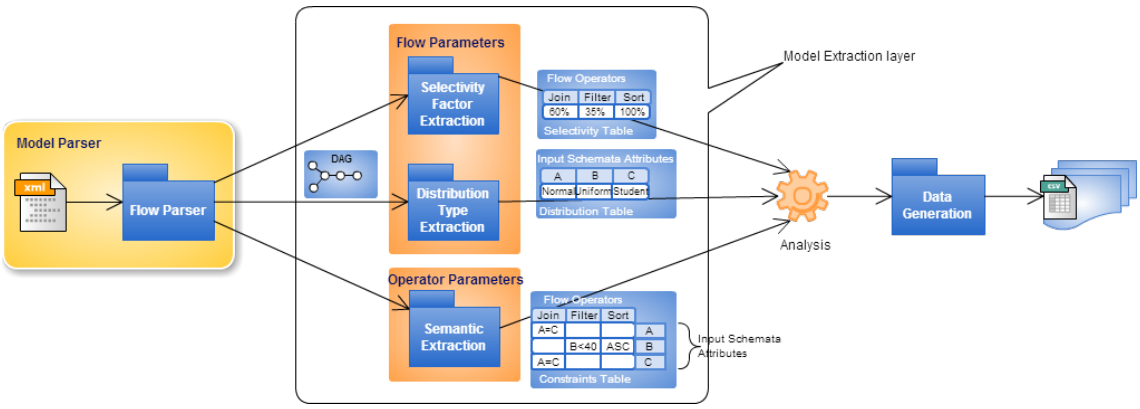



Figure 5.7: Data generation process flow

1. Model Parsing

   The process starts with parsing the ETL model (xLM) implemented by the Model Parsing layer (see Figure 5.3), and transforming it into a DAG. In the figure this comprises the left-side component called Model Parser.

2. Extraction of Flow Semantics and Parameters

   Next, the process continues with the extraction phase. It consists of three tasks executed in parallel. The DAG created as a result of parsing the ETL model, is the source information for the Model Extraction (see Figure 5.8) which extracts operation semantics and configuration parameters (i.e., operation selectivity factors, distribution, datatype). The extracted information is then internally stored and sent to the analyzer component.

3. Analysis

This phase is responsible for analyzing the extracted constraint semantics and parameters. The aim is to identify the specific behavior that generated data should follow, which is later used in the data generation algorithm.

4. Data Generation The analyzed information regarding flow parameters as well as flow semantics are the source for the data generation phase, which uses it in order to produce the data that follow and comply to the analyzed semantics.

5. Output

   The final produced output is then presented to the end-user via the user interface communication.

### 5.2.2 Algorithm

The algorithm (see Algorithm 1 and 2) explores the input logical model of an ETL process (ETLFlow), extracts the flow constraints, as well as the generation parameters at the level of attributes and ETL operations and generates the data led by the extracted parameters.

In particular, the algorithm includes three main stages (i.e., (1) extraction, (2) analysis, and (3) data generation), realized respectively within three different layers of the framework architecture (i.e., (1) Model Extraction, (2) Model Analysis, and (3) Data Generation; see Figure 5.1).

Before going into the details of these three steps of the algorithm, we present the main structures maintained by the algorithm. While analyzing the given ETL process model, we keep three structures for recording different parameters used during the data generation stage.

1. Attribute parameters (AP)

   It is an array that retains the data generation parameters at the level of individual attributes of input datastores of an ETL process (see Figure 5.9). An element of this array contains information about the considered (used) attribute (i.e., attribute name, attribute datatype, attribute property list).

Attribute property list further contains an extensible list of attribute properties that are considered during data generation (e.g., *distribution = uniform*).

2. Operation parameters (OP)

   An array that holds information about the data generation parameters at the level of operations of the input ETL process model (see Figure 5.10). An element of this array contains information about the considered ETL operation (i.e., operation name, operation property list). Operation property list further contains an extensible list of operation or quality properties that should be considered during data generation (e.g., *operation_selectivity = 0.37*).

3. Constraints Matrix (TC )

   This is a two-dimensional array structure (see Figure 5.11) that for each attribute (rows) of the input datastores, and each operation (columns) of the input ETL process, contains a set of constraints that the given ETL operation applies over the given input attribute.

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

1. *Extraction* stage (see Algorithm 1) starts from the logical model of an ETL process (*ETLFlow*). We obtain the source data stores from the process DAG (Step 3). The algorithm then for each attribute of the source data stores (i.e., *a[i]*; Step 6) and each operation following the topological order of the *ETLFlow* (i.e., *o[j]*; Steps 10, 13) extracts the data generation parameters, (i.e., Steps 7 and 11), respectively. At the same time, this stage extracts the semantics of each operation *o[j]* and searches for the constraints that the operation applies over the given attribute *a[i]* (i.e., *c[i,j]*; Step 12). As a result, extraction stage generates the above mentioned structures (i.e., AP, OP, and TC) used through the rest of the approach.

**Algorithm 1** ETL Data Extraction Algorithm

**Input:** ETLFlow

**Output:** AP, OP, TC

```
1:  procedure DATA EXTRACTION
2:      AP ← ϕ; OP ← ϕ; TC ← ϕ;
3:      𝔻𝕊 ← SourceNodes(ETLFlow);
4:      for each DS ∈ 𝔻𝕊 do
5:          SI ← InputSchema(DS);
6:          for each attribute a[i] ∈ SI do
7:              AP[i] ← Extract(a[i]);
8:              j ← 0;
9:              o_prv ← TopologicallyFirst(ETLFlow);
10:             while (hasTopologicallyNext(ETLFLow,o_prv)) do
11:                 OP[j] ← Extract(o[j]);
12:                 TC[i,j] ← Extract(c[i,j]);
13:                 o[j] ← topologicallyNext(ETLFlow,o_prv);
14:                 o_prv ← o[j];
15:                 j++;
16:             end while
17:         end for
18:     end for
19: end procedure
```

**Algorithm 2** ETL Data Analysis and Generation Algorithm

**Input:** AP, OP, TC, size

**Output:** GenData

```
1:  procedure DATA ANALYSIS AND GENERATION
2:      visited ← Boolean Array[Attributes(TC)] {false};
3:      for (i := 1 to Rows(TC)) do
4:          if (!visited[i]) then
5:              visited[i] ← true;
6:              genParams ← ϕ;
7:              SetRange(range_i,defaultBoundaries(datatype_i));
8:              gP_i ← Analyze(AP[i]);
9:              for (j := 1 to Operations(TC)) do
10:                 Update(gP_i, Analyze(OP[j]));
11:                 Add(genParams, gP_i);
12:                 for each k ∈ DependentAttributesIndexes(TC[i,j]) do
13:                     visited[k] ← true;
14:                     SetRange(range_k,defaultBoundaries(datatype_k));
15:                     gP_k ← Analyze(AP[k]);
16:                     for (l := 1 to Columns(TC)) do
17:                         Update(gP_k, Analyze(OP[l]));
18:                         UpdateRange(range_k,TC[k,l]);
19:                         UpdateRange(range_i,TC[k,l]);
20:                         if (isSelectivityRequired) then
21:                             UpdateRangeInverse(range_kInv,TC[k,l]);
22:                         end if
23:                         Add(genParams, gP_k);
24:                     end for
25:                 end for
26:                 UpdateRange(range_i,TC[i,j]);
27:                 if (isSelectivityRequired) then
28:                     size_1 ← Calculate(OP[j],size);
29:                     size_2 ← CalculateInverse(OP[j],size);
30:                     UpdateRangeInverse(range_iInv,TC[i,j]);
31:                 end if
32:             end for
33:             for each gP_i ∈ genParams do
34:                 GenDataPass ← GenerateData(gP_i,range_i,size_1);
35:                 GenDataInverse ← GenerateData(gP_i,range_iInv,size_2);
36:                 GenData ← Union(GenDataPass,GenDataInverse);
37:             end for
38:         end if
39:     end for
```

2. *Analysis* stage (see Algorithm 2) is responsible for iterating over each attribute of the generated structures, analyzing how the collected parameters (i.e., AP and OP; Steps 8 and 10) affect our data generation process. For each attribute (i.e., $i^{th}$ row of TC), we store the information used during the data generation stage (e.g., datatype, attribute properties, value ranges, etc.) inside the $gP_i$ structure. In a typical scenario, a single ETL operation may apply constraints over multiple attributes from the input. Thus, the data for these *dependent attributes* (i.e., the attributes included in the same ETL operation constraint) must be simultaneously generated. To this end, after analyzing data generation parameters of a single attribute for a single operation, we must follow the list of all *dependent attributes* from the given operation (Step 12), and analyze data generation parameters for these attributes in the same manner (Steps 15, 17, and 18). Similarly, we analyze operation constraint semantics. Based on the operation constraints, we find the range (lower and upper limit) of each attribute value (and dependent ones) and update it accordingly whenever the same attribute is encountered in the following operations (Steps 18, 19 and 26). The idea of ranges has a broad spectrum of applicability, because it can be applied to numerical attributes as well as data and textual ones. Later, these ranges will drive the data generation stage. At the end of this stage, the *genParams* list contains the information for all the dependent attributes, i.e., the attributes for which the data should be simultaneously generated.

3. *Data generation* stage (see Algorithm 2), finally, uses the generation parameters (*genParams*), resulted from the analysis stage and the ranges information, and generates data to satisfy all the restrictions extracted from the input ETL process (*ETLFlow*), (i.e., Step 36). As discussed before, data generation process can be further parameterized with additional information (e.g., the scale factor of the generated dataset - size). More details are provided in the next section.

### 5.2.3 Algorithm Illustration

To illustrate the functionality of our data generation framework, we introduce a running toy example (see Figure 5.10) that shows a simple ETL process which matches the first and last name of the customers older than 25 and loads the initials and a surrogate key to the data warehouse. The example includes several ETL operations. After extracting data from two sources ($I_1$ and $I_2$), the data are matched with the equi-join (PKey = FKey). Furthermore, the input set is filtered to keep only the persons older than 25 years (Age > 25). The first and the last name of each person are then abbreviated to their initials and the unnecessary attributes are projected out. Lastly, the data are loaded to the target data store. Thus, the algorithm we introduce, follows the topological order of the process DAG nodes, (i.e., $I_1$, $I_2$, Join, Filter, Project, Attribute Alteration, and Load) and extracts the found flow constraints (e.g., Age > 25 or PKey = FKey). Finally, data generation algorithm generates the data that satisfy the given constraints and simulate the execution of the process.

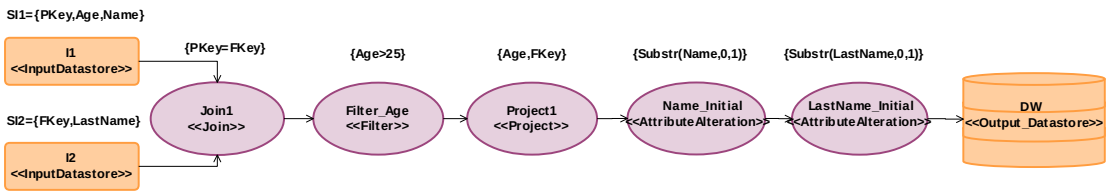



Figure 5.8: ETL flow example

Semantics of this example are given below:

- Two input datastores $\mathbb{I} = \{I_1, I_2\}$, with schemata $SI_1 = \{PKey, Age, Name\}$ and $SI_2 = \{FKey, LastName\}$.
- Schemata arity: $|SI_1|_a = 3$ and $|SI_2|_a = 2$.
- Topological order of operations is {Join, Filter, Project, Attribute Alteration, Attribute Alteration}.

1. Extraction phase

   The process (see Algorithm 1), starts with extracting parameters and constraints semantics.

First, we extract parameters at the attribute level. This is achieved by iterating over each datastore ($I_1$ and $I_2$) schemata ($SI_1$ and $S_2$) from the given ETLFlow (Figure 5.8) and extract attribute parameters. The extracted information populates the AP structure (Step 7) as per Figure 5.9.

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

Figure 5.9: Attribute parameters structure, AP

Secondly, we iterate over each operation of the flow and populate the OP structure (see Figure 5.10) with the operation parameters (i.e., selectivity) depicted in line 11 from the extraction algorithm.

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

Figure 5.10: Operation parameters structure, OP

Lastly, for each operation and each attribute from the input schemata we keep the operation semantics in the TC structure (line 10) as per Figure 5.11.

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

Figure 5.11: Table of constraints, TC

2. Analysis phase

   Next, we proceed to the analysis phase (see Algorithm 2), during which we analyze both extracted parameters and constraints semantics. As already mentioned during the explanation of the analysis stage, we analyze each attribute's parameters ($i^{th}$ row of AP) such as: datatype, precision, distribution etc., (Step 8). Similarly, we analyze operations' parameters (Step 10). The $gp_i$ structure retains these analyzed information. Moreover, in the typical ETL scenario operation semantics can be complex and applied over multiple data (i.e., several attributes present in the same ETL operation constraint). Hence, we treat these cases of *dependent attributes* together in order to generate them simultaneously. Therefore, we proceed in analyzing also the parameters of all *dependent attributes* in the same manner as we did with the single ones (i.e., only one attribute included in the ETL operation constraint), as per steps 15, 17 and 18. Since we generate all these dependent attributes altogether in one step, we keep their resulting analyzed information in the same $gp_i$ structure.

   Following the same idea, we analyze operation parameters (OP structure). For the purpose of our example, the operation parameter is the selectivity factor. Thus, at the end of our analysis we produce the exact number of values that we need to generate for each operation in order to satisfy its selectivity. For example, if an operation has a selectivity of *0.6* and the workload to be generated is *100* (i.e., size which is given by the end-user), we calculate that

60 out of 100 tuples need to satisfy the constraints of the operation, while the rest, 40, should not pass it. These two values obtained (60 and 40) are stored in the two variables $size_1$ and $size_2$ (Steps 28 and 29). In addition, we keep the inverse of operation constraints (Steps 21, 30) which is responsible for generating the tuples that should not pass the operation constraints. This information is very valuable to our data generation and is thus retained in the same generation parameters structure $gp_i$, which cumulatively collects the analyzed information of flow parameters (at the attribute and operation level).

In the same manner, we analyze operation constraints of all *dependent attributes*. Based on the operation constraints, we find the range (lower and upper limit) of each attribute value (and dependent ones) and update it accordingly whenever the same attribute is encountered in the following operations (Steps 18, 19 and 26). The idea of ranges has a broad spectrum of applicability, because it can be applied to numerical attributes as well as data and textual ones. Later, these ranges will drive the data generation stage.

In addition, since we generate dependent attributes altogether, in order to keep track of these generation process and to optimize our generation procedure, we keep a flag of all generated attributes in the visited structure (Boolean array list). At the beginning it is initialized with *false* values (Step 2), but later on is updated to *true* (Steps 5 and 13) whenever we analyze and generate an attribute and the list of the dependent ones.

3. Data generation phase

   Finally, having collected all the information from the analysis phase, we can now proceed to the data generation procedure (see Algorithm 2). It takes as input the $gp_i$, data generation parameters collector, the load size (i.e., $size_1$ and $size_2$) from the *OP* analyzer and the ranges from the constraints analyzer, and generates data simultaneously for all dependent attributes based on their respective resolved parameters and constraints.

Our algorithm works for two data generation cases: (1) when selectivity is not required and (2) when satisfying operation selectivity is required. We start giving

an example of the first case and then proceed to the second case.

Inputs: TC (see Figure 5.11), AP (see Figure 5.9), load size provided by the end-user (100).

**First Case**

This is the general case where we follow the algorithm to generate data to satisfy attribute parameters and operation constraints. However, we do not consider operation parameters (i.e., selectivity factor). This case will be discussed later.

1. First Iteration

   In the first iteration we extract and analyze the operation semantics for the first row (attribute) from *TC*. For the *PKey* attribute we find the constraint "*PKey* = *FKey*". When we analyze it, we see that it contains a dependent attribute *FKey*. Therefore, we iterate not only over the operation semantics that use *PKey*, but also those that use *FKey* (see Figure 5.12).

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

Figure 5.12: First iteration

   We find that for *FKey* there is another operation *Project* that uses it, but we do not take this into consideration since *Project* does not imply any changes to the values of the input dataset, rather it is at the schema level.

   In addition, apart from collecting operation constraints, we also extract and analyze parameters information for both attributes. The parameters we collect in this case come from *AP* and contain information that *PKey, FKey* are both numerical values with uniform distribution. Moreover, the user provides us with the workload size to be generated (i.e., 100 tuples).

| | Generated Data | | | |
|---|---|---|---|---|
| PKey | 45 | 125 | 97 | 3 |
| Age | | | | |
| Name | | | | |
| FKey | 45 | 125 | 97 | 3 |
| LastName | | | | |

Figure 5.13: First iteration result

Finally, after we collected all information required, we proceed to the data generation stage, which in our case determines the generation of 100 numerical long values that should be equal among them. The generated data up to this iteration is depicted in Figure 5.13.

Now that we generated data for *PKey, FKey*, we assign the visited flag to true for both fields (line 5,13), in order to guarantee that we will no longer iterate again over them. The rest of attributes will be populated in the following iterations.

2. Second Iteration

   During the second iteration, we proceed to the next attribute (row) from the *TC* which is *Age* (see Figure 5.14).

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

Figure 5.14: Second iteration

We collect all information about operations that use this attribute, i.e., *Filter* with the semantics "*Age >25*" and *Project*. This means that we update the range of *Age* (line 26) and set the lower limit to 25, since we need to generate values greater than 25.

Similarly to the previous case, we do not consider projection during the data generation phase since it does not impose any constraints over the values of the data. In addition, the parameters we extract regarding this attribute (see Figure 5.11) are that *Age* is an integer whose values should be normally distributed with mean 30 and variance 5. From the *TC* we do not encounter any other operation that actually used this attribute, hence we proceed to the data generation step. The parameters of the data generation should be that the value generated is a natural number with normal distribution and greater than 25.

Thus, we populate the second row of our output structure with 100 random generated values greater than 25, following that particular distribution function. A possible generated result up to this point is depicted in Figure 5.15.

| | Generated Data | | | |
|---|---|---|---|---|
| PKey | 45 | 125 | 97 | 3 |
| Age | 53 | 34 | 26 | 44 |
| Name | | | | |
| FKey | 45 | 125 | 97 | 3 |
| LastName | | | | |

Figure 5.15: Second iteration result

We mark this field as visited.

3. Third Iteration

We run the same algorithm for the next attribute, *Name* (see Figure 5.16).

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

Figure 5.16: Third iteration

The only condition we find is that there is an *Attribute Alteration* operation in the flow, that changes the value of the original by retaining only the first initial of the Name. From such constraint we conclude that the text inside the attribute *Name* should not be empty. In addition, from *AP* (see Figure 5.9) we conclude that generated values should be textual data.

Since, our data generator generates synthetic but realistic workloads, we make use of real data dictionaries in order to populate fields of e.g., Names, Countries, Cities etc.

So, particularly for this case, we generate 100 real names extracted from the *Names* dictionary we provide in our data generator framework. In case we are provided also with the information of the length of *Name* attribute at the AP, then we make sure to extract only those names from the dictionary that comply to the required field length. Finally, at the end of this step, another row will be populated from our result set, which will look as per Figure 5.17.

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

Figure 5.17: Third iteration result

4. Fourth Iteration

   The fourth iteration is also the last one. We do not consider *FKey* which is already generated from the first step (i.e., visited flag is true), so we go directly to the last attribute *LastName*, see Figure 5.18.

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

Figure 5.18: Fourth iteration

For *LastName* we find an *Attribute Alteration* operation with the exact semantics as for the *Name* attribute. Hence, we follow the same procedure to generate 100 realistic last names from the dictionary of *Last Names*.

Finally, the result dataset is complete.

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

Figure 5.19: Fourth iteration result

5. Generate ETL data

   This is the final step where we present the final output to the user (i.e., csv format). We generate as many result datasets as there are input relations in our ETL flow (i.e., two datasets for $I_1$ and $I_2$).

   As we introduced the TC (table of constraints), it contains one row for each of the attributes of the input schemata. Hence, in order to extract the portion of data related to only one input schemata, we need to horizontally split the generated output.

   From the result set we created (which contains data for both input datasets) we split it based on the schemata of $I_1$ and $I_2$. From the information we parsed from the ETL flow (see Figure 5.8), schema of $I_1$ is $SI_1$ = {PKey, Age, Name} with arity $|I_1|$=3 and schema of $I_2$ is $SI_2$ = {FKey, LastName} with arity $|I_2|$=2. So we have to horizontally partition our output into two different sets exactly after the third row.

   The first upper partition contains valid generated data for $I_1$, that satisfies the ETL flow (Figure 5.8), while the bottom one contains the generated data for $I_2$. Each entry (tuple) of the input relation is a vertical combination of values from the same column, which means that we need to transpose the result set to finally be able to present it to the user in a suitable format (i.e., csv). Refer to Figure 5.20 for illustration.

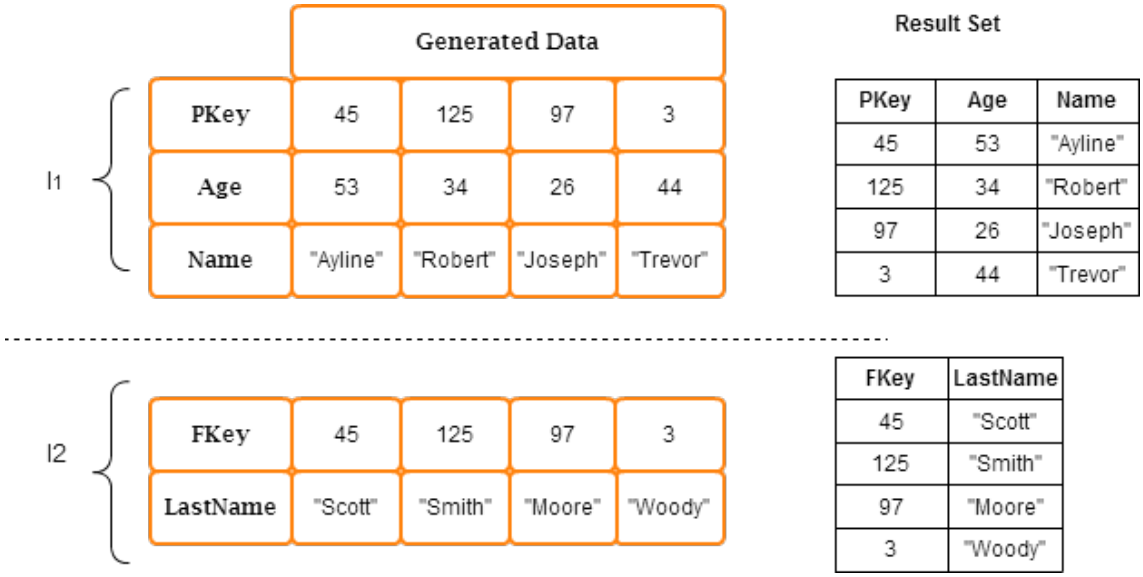



Figure 5.20: Final result

**Second Case**

The above description was the general case of data generation without considering the selectivity factors. But, given that our data generator aims at generating data to satisfy other configurable parameters, we illustrate in this second example the adaptability of our algorithm to the problem of generating data to not only satisfy ETL flow semantics but also other parameters (i.e., operation selectivity). Hence, the algorithm applied is the same, with the difference that now we also consider the parameters extracted and retained at the *OP*.

In what follows, we give the insights of generating data to satisfy selectivity factor for the *Filter* operation solely. We proceed the exact way as the above case, meaning we iterate row by row over the TC and over the operation constraints of *Filter*. The difference is that now we also extract and analyze the operation parameters from *OP*.

From the OP (see Figure 5.10) we find that *Filter* operation has a selectivity of *0.7*. While iterating over the TC, we extract operation semantics and notice that *Filter* operation uses attribute *Age*. For this operation we find the constraint "*Age* > *25*". With the selectivity factor of *0.7* from OP, we conclude that out of all incoming tuples for the Filter, 70% will satisfy its constraints (*Age* values should be greater than 25), while 30% will not (*Age* values should be smaller or equal to 25).

**Analysis of selectivity**

- To determine the total number of incoming tuples for *Filter*, we consider preceding operations, which in our case is *Join* with selectivity *0.6*. This means that in total 0.6*(100×100) = 6,000 tuples pass the join condition.
- From these 6,000 tuples only 70% (as per *Filter* selectivity), which means 4,200 will successfully pass the filtering condition ("$Age > 25$") along with the join one ("$PKey = FKey$").
- The remaining of 1,800 should fail ("$Age \leq 25$"). In order to generate the data that do not pass this operation of the flow, we rely on the inverse constraints that we parse from the algorithm (Steps 21, 30).

Finally, after we collected and analyzed information from TC ("$Age > 25$"), AP (long value normally distributed with mean 30 and standard deviation 5) and OP (selectivity *0.7*), we proceed to the data generation phase. Similarly, since *Join* operation proceeds the *Filter*, we consider its semantics also ("$PKey = FKey$"). Its respective parameters suggest long numerical values having a uniform distribution.

As a result of the above analysis, we need to generate a dataset ($I_1$ and $I_2$) such that the output of *Join* operation is 6,000 tuples that satisfy join condition, out of which 4,200 have *Age* greater than 25, while the rest have *Age* smaller or equal to 25.

# 6 PROTOTYPING

In this chapter we introduce the technical details of the ETL data generation prototype. First of all, we introduce the technological environment used to develop and test the prototype along with the basic data structures used and the functionality they provide. Secondly, we test the performance of our data generator by running different experiments and measuring the time taken to generate the data (i.e., when changing load size, flow complexity). Finally, we show the results of our testing in the experimental results subsection and conclude with a short discussion of our findings.

The development approach is a resemblance of the "*Agile methodology*", since our approach was incremental. We started the implementation from a base set of ETL operations, and then incrementally added new functionalities to support new operations and more complex expressions. As soon as the current task was finished, we moved on to the next one and hence extended and improved the code implemented. It is important to mention that the approach followed is a variation of the agile methodology since we did not have strictly defined sprints, though we had regular meetings and milestones to be achieved on a weekly basis.

## 6.1 Implementation

In this section we introduce the technological environment and technical details of used data structures.

In the implemented prototype we focus on a minimal set of ETL operations form

the list defined in the previous chapter. Specifically, we implement an ETL data generation prototype for four operations: *Join, Filter, Project, Attribute Addition.* Moreover, we give support to additional parameters such as attribute characteristics and operation parameters e.g., selectivity.

### 6.1.1 Technologies Used

In this section we present the technological environment used for developing and testing the prototype.

#### Programming Language

This research is part of an ongoing project at the group for Information Modelling and Processing (MPI) at UPC. As such, the best strategy would be to continue working under the same environment so that the integration of each separate project would be more feasible and less error-prone. Another strong reason is that this prototype reuses part of the existing codes and projects developed by other colleagues, such as the Model Parsing layer of the ETL data generation architecture described in the previous chapter. Hence, for interoperability and portability reasons we continued working with Java technologies (platform and programming language).

Java is an object-oriented programming language. It offers strong capabilities to design reusable code wrapped in classes that can be run in many parts of the java application. This improves the modularity of the code and moreover, it offers extensibility capabilities in order to extend existing classes with additional functionalities. In addition, Java code can run on any Java Virtual Machine and its popularity allows for the use of already implemented libraries.

#### Development Platform

The prototype has been developed using Eclipse Java IDE for Web Developers Indigo Service Release Version. Eclipse is an integrated development environment (IDE) which offers an extensible plug-in system for customizing the environment.

### 6.1.2 Implemented Architecture

In this section we introduce the implemented architecture and the corresponding components, reused or designed.

#### Model Parsing Layer

We implement the model parsing layer (see Figure 5.3), as an external source to our implementation, because we reuse the *ETLFlowGraph* package (see Figure 6.1) that was already implemented as part of the existing project at the group for Information Modelling and Processing (MPI).

*ETLFlowGraph* implements both, the *File Parser* and the *File Handler* component of the model parsing layer. *File Handler* manages the communication between the *File Parser* and the external API and user interface layer, whereas *File Parser* is responsible for transforming the xLM file (XML representation of the ETL model) into a DAG. In Figure 6.1, we have extracted an excerpt from the UML diagram of the *ETLFlowGraph* package and the corresponding classes reused in our prototype.

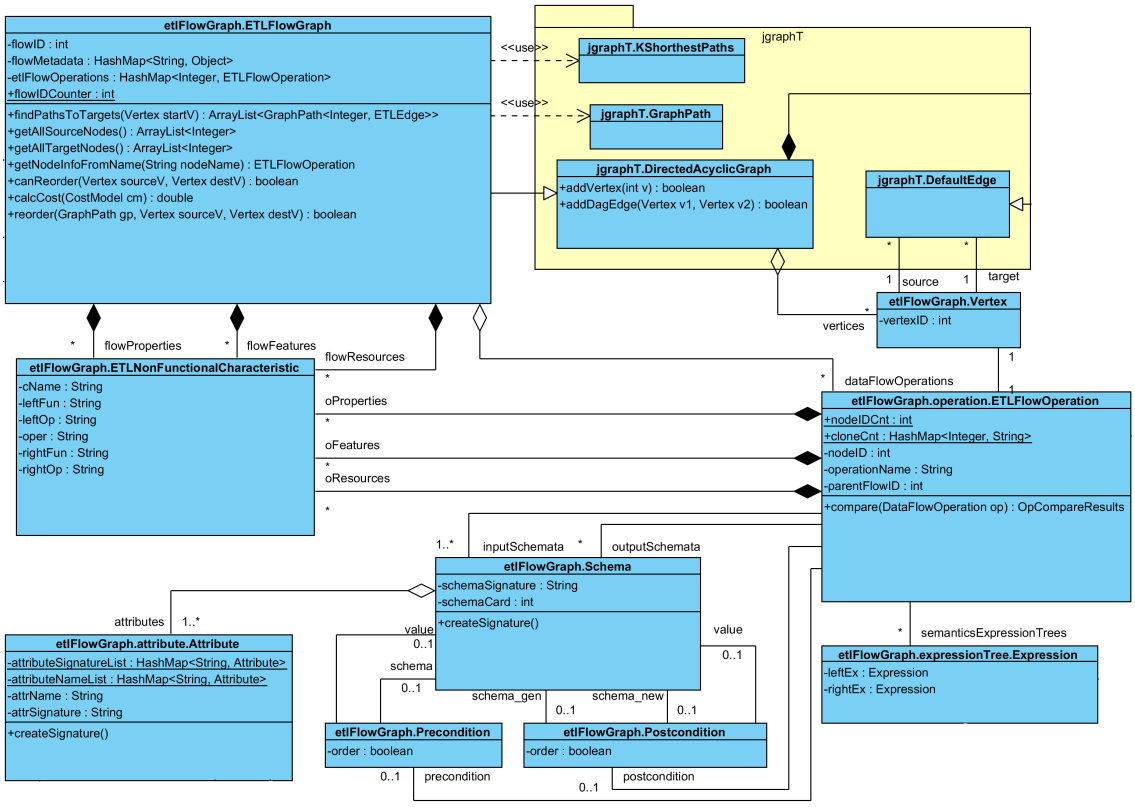



Figure 6.1: Referenced architecture

We list below the classes we reused from the *ETLFlowGraph* package, along with a brief description of the functionalities useful to our implementation.

- ETLFlowGraph class

  This class is an extension of the "*DirectedAcyclicGraph*" class from the "*jGraphT*" library in Java. It takes an XML as input (the xLM representation of the ETL model) and returns a DAG, where nodes represent ETL entities (activities, data stores and corresponding schemata) while edges represent the data flow from the source to the target entity. We use this component to track input sources of the entire ETL flow, as well as each operation separately. We also use it in order to retrieve iteratively the operations (nodes of the graph) to be able to later process them for retrieving operation properties e.g., name and type of the operation and the corresponding semantics.

- ETLFlowOperation class

  The operation class allows us to extract operation properties for each of the operations retrieved from the *ETLFlowGraph* class. An example of operation properties that we extract is the *Operation type*, which might be a *Datastore* or *Operation*.

- ExpressionTree class

  *ExpressionTree* is the class that retrieves the behaviour of the operation, which is expressed by its semantics. This class expresses the semantics as a tree, where the internal nodes contain an algebraic operator (i.e., +, –, /, *, etc.) or attribute used during the operation. In Figure 6.3, we illustrate a simple mathematical expression such as: (a + b) * (12 - c) expressed as a tree.

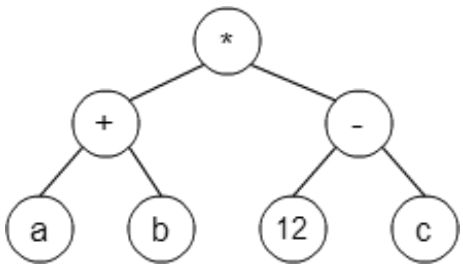



Figure 6.2: Example of an expression tree

- Attribute class

  The *Attribute* class keeps information about the attributes used in the operation and their corresponding properties i.e., attribute name, datatype, precision. We make use of this class in order to retrieve information about the attributes that were consumed in each operation expression.

- Schema class

  *Schema* class is used to define the input and output schema of each operation. This knowledge is important for tracking the schema changes due to applying ETL operations semantics in the flow and it is also taken into consideration for the data generation process.

### Model Extraction Layer

The rest of the architecture has been implemented following the UML diagram presented in Figure 6.3.

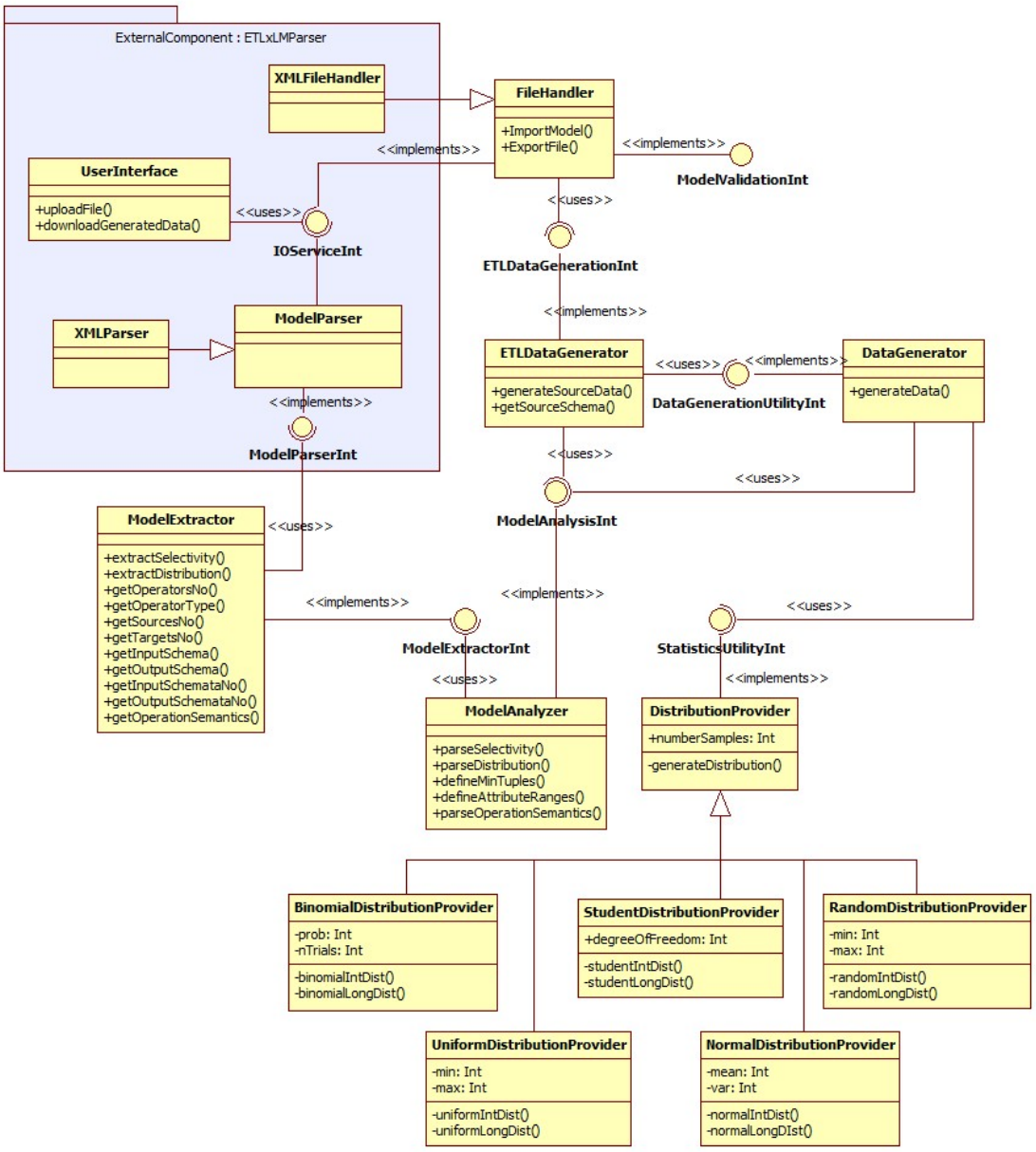



Figure 6.3: Implementation architecture

The *Model Extraction* layer (see Figure 5.4) is implemented by the *Model Extractor* interface which is responsible for extracting both operation semantics and model parameters. Parameters we extract are at the attribute level e.g., attribute datatype, or at the operation level e.g., operation selectivity factor. Operation selectivity factor can be defined as the proportion of input data that satisfy the constraints of the operation and are passed to the output dataset. For example, if the selectivity factor of a *Filter* operation is *0.6*, this means that *60%* of the incoming tuples satisfy the

flow and are present also in the resulting dataset.

To retain the extracted information about model parameters we create a specific data structure that we refer to while generating the data.

Similarly, we create another structure that we call Table of Constraints (TC) introduced in the algorithm section from the previous chapter, which serves the purpose of systematically storing the semantics extracted above. To recall, from the implementation point of view, it is a two-dimensional list in which each cell of the matrix holds operation semantics and corresponding consumed attributes by each operation. The object position in the table is relevant, since the row stands for the attribute being consumed in the operation belonging to that particular column (see Figure 5.13). The row dimension has a size equal to the sum of schema cardinality of all input datasets, whereas the second dimension has a size equal to the number of operations present in the flow. However, in case there are operations that add new attributes to the initial datastores' schemata, then we extend the TC by appending a new row, in order to also consider these additional attributes and their respective constraints during the data generation process.

### Model Analysis Layer

The *Model Analysis* layer (see Figure 5.5) is implemented by the *Model Analysis* interface, which analyzes the information extracted by the extraction layer. The information we analyze is related to operation semantics and other model parameters. The analyzed information is relevant to generate data that should satisfy the operation constraint semantics and additional flow parameters. As a result, we guarantee a data generation prototype that simulates the behaviour of a realistic ETL flow.

### Data Generation Layer

The *Data Generation* layer (see Figure 5.6) is implemented by the *Data Generation Utility* and *ETL Data Generation* interfaces. The *Data Generator Provider* component is implemented by the *Data Generator* class that realizes the *Data Generation Utility* interface, while the *Data Generator Provider* component is implemented by the *ETL Data Generator* class that realizes the corresponding *ETL Data Gener-*

*ation* interface. We generate numerical and textual data based on the datatype information extracted by the extraction layer and the semantics (expression tree) parsed and analyzed for each operations.

In our architecture we also provide data generation based on other attribute parameters such as distribution function. Hence, we introduce in our class diagram the *Statistics Utility* interface which communicates with the *Data Generator* in order to provide the data generation functionality of generating numerical data that need to respect a particular statistical distribution function. To this end, we make use of Math Java libraries.

## 6.2 Experimental Setup and Testing

Testing is an important part of the software development cycle which aims at verifying that the implemented software behaves according to the initial requirements and expectations. We focused on two types of testing:

- Functional testing

  It aims at verifying that the software produces the desired output and performs the expected actions. This type of testing can focus on the entire software functionality or specific features and components. We performed functional testing along the complete development life-cycle.

- Non-functional testing

  It aims at measuring features of the software other than functional capabilities such as: performance, scalability, security. It determines the quality of the implemented software. We performed performance testing on several ETL test cases that we describe below.

Our experiments were carried under a Windows 32-bit machine, Processor Core 2 Duo, 2.1 GHz and 4GB of RAM. The test cases contain all the ETL operations that we have implemented in our prototype. We start with a basic flow containing: *Join, Filter, Attribute Addition* and *Project* operation. Starting from this basic

scenario, we create others more complex ones, in which we incrementally add more operations. The motivation comes from the fact that obtaining the real world set of ETL flows covering different scenarios with different complexity and load sizes is hard. Therefore, we implemented a functionality that replicates the operations of the existing flow and adds them to the same flow incrementally making the flow more complex. In what follows we demonstrate this procedure.

**Basic scenario**

The basic scenario contains two input datastores $I_1$, $I_2$, and the considered operations are *Join, Filter, Project, Attribute Addition*. So in total, we have four operations present in the flow. Figure 6.5 illustrates it graphically.

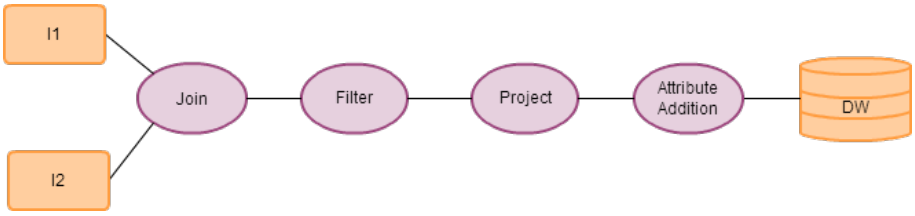



Figure 6.4: Basic ETL scenario

**Scenarios creation iteration**

Starting from this basic scenario, we create more complex ETL flows by adding additional operations, i.e., *Join, Filter* in various positions of the original flow. Figure 6.6 depicts the formation of new flows starting from the basic one.

*It should be noted that, when adding another Join operation to the flow, we add also an Input Datastore and a Project in order to guarantee the correctness of the new created flow. For our experiments, the newly added Input Datastore is a copy of an existing one that precedes the point of application, whereas the Project succeeds the Join operation.*

We collect execution times for 6 cases, starting from a basic ETL flow and continuing to more complex ones:

1. Case 1 – Basic ETL scenario, consisting of four operations implemented, i.e., *Join, Filter, Project, Attribute Addition* as described above.

2. Case 2 – ETL scenario consisting of 5 operations, originating from the basic one by adding an additional *Filter* operation to the flow.

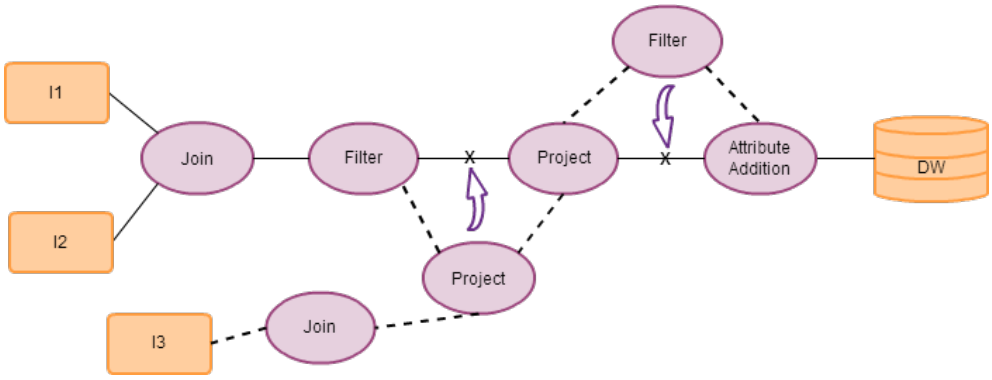



Figure 6.5: Incremental ETL scenario

3. Case 3 – ETL scenario consisting of 6 operations, originating from the basic one by adding either two additional *Filter* operations, or a *Join* operation to the flow. To recall, when adding a *Join* operation we also add a *Project* and an *Input Datastore* in order to guarantee matching schemata. However, given that we consider only transformation operations during the generation algorithm, the added *Input Datastore* does not impact the generation procedure, hence only the added *Join* and *Project* are relevant for us. The position where these operations are added to the flow is not predefined, rather it is randomly assigned at the execution time.

4. Case 4 – ETL scenario consisting of 7 operations. Additional *Join* and *Filter* operations are added to the basic flow, randomly assigning the added position on the fly.

5. Case 5 – ETL scenario consisting of 8 operations, which is derived from the basic scenario by adding *Join* and two *Filter* or two additional *Join* operations.

6. Case 6 – ETL scenario consisting of 9 operations. Two additional *Join* operations along with a *Filter* operations are added to the basic flow.

## 6.3 Experimental Results

We measure the execution time of the data generation process for the above mentioned 6 cases of ETL flows. For each given ETL scenario we generate 4 different

datasets (load size). We measure the load size in number of generated tuples per each input datastore of the flow.

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

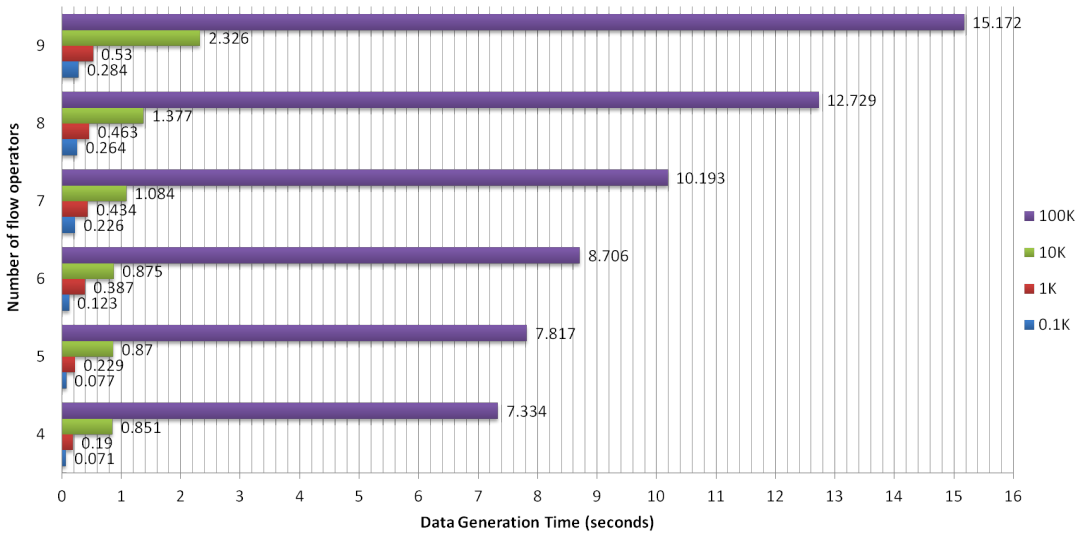



Figure 6.6: Generation time wrt flow complexity

Figure 6.6 illustrates the increasing generation time when moving from the simplest ETL scenario to a more complex one while keeping the load size constant. In addition, it also shows the increasing generation time when increasing the load size from 100 until 100,000 tuples. The margin of increasing execution time is higher, as the amount of generated data increases, which indicates an exponential cost.

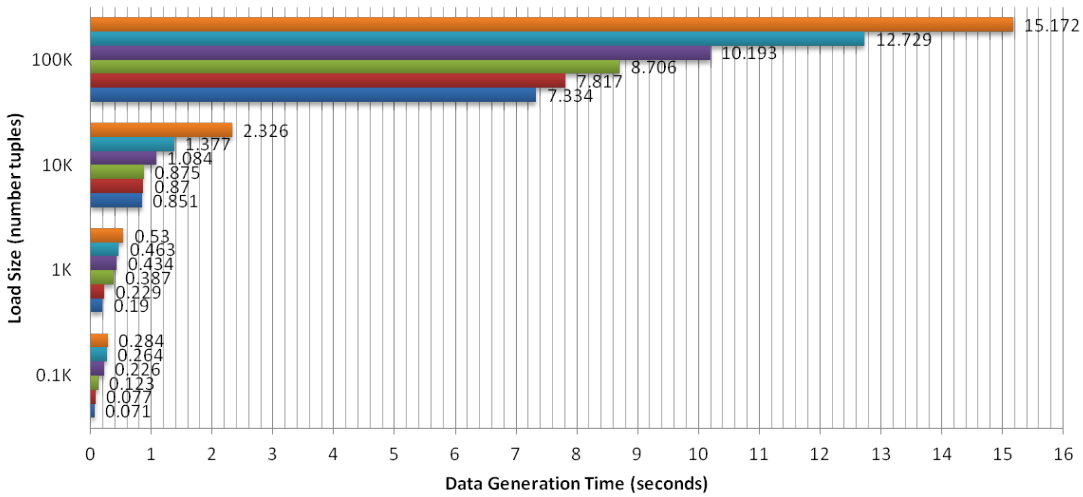



Figure 6.7: Generation time wrt load size

Figure 6.7 shows the generation time for the 6 ETL scenarios, tested under the 4 experiments sets of generating 100 until 100,000 tuples. The behaviour noticed is an increasing generation time as the load size is higher and as the complexity of the flow rises.

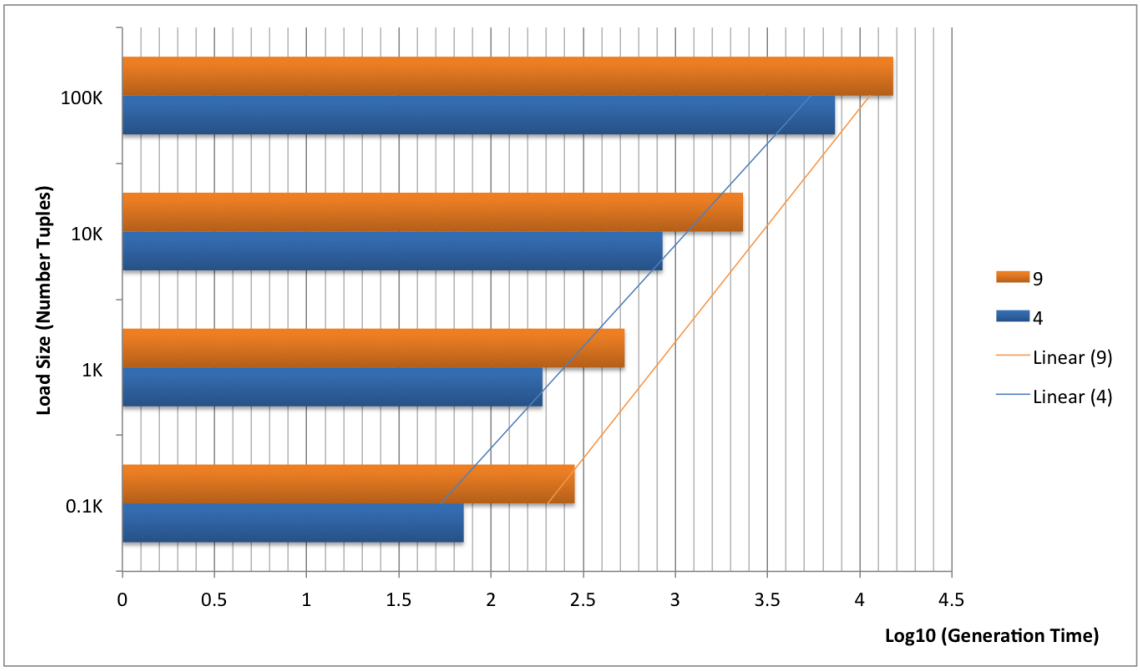



Figure 6.8: Linear trend of the data generation performance for flow with complexity 4 and 9 as load size increases

Figure 6.8 demonstrates the linear trend of the generation time as the load size increases for the two extreme cases under study: (1) for ETL flow with complexity 4 (consisting of 4 operations) and 9. In the vertical axes is depicted the varying load size expressed in number of tuples generated, from 100 until 100,000 tuples. Whereas, in the horizontal axes there is the generation time, expressed as the logarithm of base 10 of the time in milliseconds. The reason we consider the logarithm of the generation time is to proportionally scale the variation in the load size and the corresponding execution time. To be noticed is that the performance shows a linear trend, with a decreasing slope when moving from the simplest ETL flow (4) to the more complex one (9), which suggests scalability opportunities.

## 6.4 Discussion

From the set of experiments performed, we conclude that the higher the ETL flow complexity the higher the data generation time. This is justified by the fact that the semantics of the flow are increasing in number and complexity. Hence, more rules and more constraints are imposed over the generated data. Figure 6.6 illustrates graphically this observation.

The other observation is that the data generation time is also dependent on the load size. This is obvious since the more data to be generated, the more time is required to achieve it. Figure 6.7 illustrates this observation from our set of conducted experiments.

What is to be highlighted though, is that the margin of increased generation time when increasing the load size is higher than when increasing the complexity of the flow (see Figure 6.6 and 6.7). We can conclude that the resolving operation semantics and constraints is overcome by the time needed to generate the actual data. A solution to optimize it would be parallelizing the generation of independent datasets. This is supported by the other observation of the linear tendency as shown in Figure 6.8. This linear trend demonstrates a lower slope as the load size increase, which suggests that our data generation framework can be scaled up to accommodate the parallelization goals. Whereas, the indicated cost with respect to

flow complexity (see Figure 6.6) is exponential.

# 7 CONCLUSIONS AND FUTURE WORK

In this final chapter, we summarize the results and contribution of this master project. This thesis aims at proposing a semantic-aware ETL Data Generation framework that provides an automatic, smart way to generate representative ETL data for simulation, testing and benchmarking purposes.

## 7.1 Contribution

The contribution of this thesis lies in automatically providing testing workloads for data-centric processes which is semantic-aware and parameterized in many levels.

We provide the generation of a common workload necessary for testing data-centric processes (i.e., ETL processes) which is important to guarantee a correct process that delivers the right information to the user. However, besides correctness, the information delivered to the end-user should also meet other quality criteria (e.g., reliability, recoverability, freshness, etc.) to ensure data is delivered in an efficient way. However, providing such data to test the fulfillment of all these quality criteria is often difficult due to confidentiality issues, expensive transfer over the network, complexity etc. This is mostly due to the fact that a single dataset usually does not represent the evolution of data throughout the complete process lifespan, and hence it cannot cover the plethora of all possible test cases. Moreover, the complexity of the required data is such that it needs to simulate the behavior of multiple scenarios that take many parameters into account which in turn is a labor intensive task.

What we propose, is an automatic data generator for data-centric processes. By

extracting and analyzing the semantics of data transformations and dependency constraints they imply over data, we automatically generate testing datasets. In addition, the framework proposed is configurable for many characteristics (e.g., distribution, selectivity) and can be extended with additional functionalities. To this end, we also contribute in proposing an ETL operation taxonomy and a formalization of ETL operations semantics definition.

We have tested the feasibility of our approach by implementing an ETL data generation prototype. From the experimental phase we show a linear behaviour of the performance of the implemented prototype, which suggests a scalable system that can accommodate more intensive tasks (i.e., high complexity ETL flows, higher volumes of workloads).

## 7.2 Future Work

Although the framework we present is complete and covers the most generic ETL operations and other important parameters (e.g., load size, distribution, selectivity etc.) still it can be extended to cover a broader range of parameters for different datasets and transformation characteristics in order to cover a variety of test scenarios. Some of these extensible features are presented below:

- Extend the list of supported operations

  As discussed in chapter 4, we consider atomic operations that are generic and found in most of the data integration tools. Also, we do not consider user defined components since they are not general but quite specific to a particular scenario. However, our framework is extensible to covering also other complex operations (expressed as a combination of atomic ones already supported).

- Support for complex predicates

  In chapter 4, we also discuss about the operation semantics and how we formalize them. In the current proposed framework, we cover simple predicates. However, the formalization we introduce has high expressiveness. Hence, it

can support the formalization of complex semantics also, which can be expressed as a complex predicate containing multiple atomic ones connected by logical operators. In addition, also our prototype can be extended since we can represent any possible predicate as an expression tree.

- Additional parameters

  The framework can be extended to cover a broader spectrum of configurable parameters, other from the ones we already cover.

Future developments might be performed also over the implemented prototype in terms of extended functionalities and optimization possibilities. The prototype developed and presented in this master thesis is an implementation to prove the feasibility of the proposed theoretical framework. Hence, it does not cover the full list of ETL operations. Therefore, it can be extended to cover other operations as well. In addition, similarly to the framework, it can be extended to support other model parameters also. Another important matter, is the opportunity to scale up the system in order to achieve higher performance as suggested by the results of the experimental work.

# Bibliography


[1] Kevin Wilkinson, Alkis Simitsis, Malú Castellanos, and Umeshwar Dayal. Leveraging Business Process Models for ETL Design. In Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson C. Woo, and Yair Wand, editors, *ER*, volume 6412 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 2010.

[2] Zineb El Akkaoui and Esteban Zimányi. Defining ETL worfklows using BPMN and BPEL. In Song and Zimányi [53], pages 41–48.

[3] Mario Barbacci, Mark H Klein, Thomas A Longstaff, and Charles B Weinstock. Quality attributes. Technical report, DTIC Document, 1995.

[4] Panos Vassiliadis, Alkis Simitsis, and Eftychia Baikousi. A taxonomy of ETL activities. In Song and Zimányi [53], pages 25–32.

[5] Timos K. Sellis. Formal specification and optimization of ETL scenarios. In Song and Vassiliadis [54], pages 1–2.

[6] Hans Peter Luhn. A business intelligence system. *IBM Journal of Research and Development*, 2(4):314–319, 1958.

[7] Arisa Shollo. *The Role of Business Intelligence in Organizational Decision-making*. Copenhagen Business SchoolCopenhagen Business School, Institut for Produktion og ErhvervsøkonomiDepartment of Operations Management, 2013.

[8] Matteo Golfarelli, Stefano Rizzi, and Iuris Cella. Beyond data warehousing: what's next in business intelligence? In Il-Yeol Song and Karen C. Davis, editors, *DOLAP*, pages 1–6. ACM, 2004.

[9] Barbara Wixom and Hugh J. Watson. The bi-based organization. *IJBIR*, 1(1):13–28, 2010.

[10] Thomas D. Clark Jr., Mary C. Jones, and Curtis P. Armstrong. The dynamic structure of management support systems: Theory development, research focus, and direction. *MIS Quarterly*, 31(3):579–615, 2007.

[11] Thomas H Davenport. Competing on analytics. *Harvard Business Review*, 84(1):98, 2006.

[12] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013.

[13] Zineb El Akkaoui, Esteban Zimányi, Jose-Norberto Mazón, and Juan Trujillo. A model-driven framework for ETL process development. In Il-Yeol Song, Alfredo Cuzzocrea, and Karen C. Davis, editors, *DOLAP*, pages 45–52. ACM, 2011.

[14] Zineb El Akkaoui, Jose-Norberto Mazón, Alejandro A. Vaisman, and Esteban Zimányi. BPMN-Based Conceptual Modeling of ETL Processes. In Alfredo Cuzzocrea and Umeshwar Dayal, editors, *DaWaK*, volume 7448 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2012.

[15] Panos Vassiliadis, Zografoula Vagena, Spiros Skiadopoulos, Nikos Karayannidis, and Timos K. Sellis. ARKTOS: towards the modeling, design, control and execution of ETL processes. *Inf. Syst.*, 26(8):537–561, 2001.

[16] Alkis Simitsis. Modeling and managing ETL processes. In Marc H. Scholl and Torsten Grust, editors, *VLDB PhD Workshop*, volume 76 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[17] Alkis Simitsis, Panos Vassiliadis, Manolis Terrovitis, and Spiros Skiadopoulos. Graph-Based Modeling of ETL Activities with Multi-level Transformations and

Updates. In A. Min Tjoa and Juan Trujillo, editors, *DaWaK*, volume 3589 of *Lecture Notes in Computer Science*, pages 43–52. Springer, 2005.

[18] Alkis Simitsis, Kevin Wilkinson, Malú Castellanos, and Umeshwar Dayal. QoX-driven ETL design: reducing the cost of ETL consulting engagements. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 953–960. ACM, 2009.

[19] Umeshwar Dayal, Kevin Wilkinson, Alkis Simitsis, Malú Castellanos, and Lupita Paz. Optimization of Analytic Data Flows for Next Generation Business Intelligence Applications. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *TPCTC*, volume 7144 of *Lecture Notes in Computer Science*, pages 46–66. Springer, 2011.

[20] Alkis Simitsis, Kevin Wilkinson, and Petar Jovanovic. xPAD: a platform for analytic data flows. In Ross et al. [55], pages 1109–1112.

[21] Juan Trujillo and Sergio Luján-Mora. A UML Based Approach for Modeling ETL Processes in Data Warehouses. In Il-Yeol Song, Stephen W. Liddle, Tok Wang Ling, and Peter Scheuermann, editors, *ER*, volume 2813 of *Lecture Notes in Computer Science*, pages 307–320. Springer, 2003.

[22] Sergio Luján-Mora, Panos Vassiliadis, and Juan Trujillo. Data Mapping Diagrams for Data Warehouse Design with UML. In Paolo Atzeni, Wesley W. Chu, Hongjun Lu, Shuigeng Zhou, and Tok Wang Ling, editors, *ER*, volume 3288 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2004.

[23] Dimitrios Skoutas, Alkis Simitsis, and Timos K. Sellis. Ontology-Driven Conceptual Design of ETL Processes Using Graph Transformations. *J. Data Semantics*, 13:120–146, 2009.

[24] Dimitrios Skoutas and Alkis Simitsis. Designing ETL processes using semantic web technologies. In Song and Vassiliadis [54], pages 67–74.

[25] Eric Thoo, Ted Friedman, and Mark A Beyer. Magic Quadrant for Data Integration Tools. *Gartner RAS Core Research Note G*, 248961, 2013.

[26] Pall Amanpartap Singh and Jaiteg Singh Khaira. A comparative review of extraction, transformation and loading tools. *Database Systems Journal BOARD*, page 42.

[27] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. Optimizing ETL Processes in Data Warehouses. In Karl Aberer, Michael J. Franklin, and Shojiro Nishio, editors, *ICDE*, pages 564–575. IEEE Computer Society, 2005.

[28] Alkis Simitsis, Panos Vassiliadis, Umeshwar Dayal, Anastasios Karagiannis, and Vasiliki Tziovara. Benchmarking ETL Workflows. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *TPCTC*, volume 5895 of *Lecture Notes in Computer Science*, pages 199–220. Springer, 2009.

[29] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. DIPBench: An independent benchmark for Data-Intensive Integration Processes. In *ICDE Workshops*, pages 214–221. IEEE Computer Society, 2008.

[30] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In Richard T. Snodgrass and Marianne Winslett, editors, *SIGMOD Conference*, pages 243–252. ACM Press, 1994.

[31] Zijian Ming, Chunjie Luo, Wanling Gao, Rui Han, Qiang Yang, Lei Wang, and Jianfeng Zhan. BDGS: A Scalable Big Data Generator Suite in Big Data Benchmarking. *CoRR*, abs/1401.5465, 2014.

[32] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. BigDataBench: a Big Data Benchmark Suite from Internet Services. *CoRR*, abs/1401.1406, 2014.

[33] Tilmann Rabl, Michael Frank, Hatem Mousselly Sergieh, and Harald Kosch. A data generator for cloud-scale benchmarking. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *TPCTC*, volume 6417 of *Lecture Notes in Computer Science*, pages 41–56. Springer, 2010.

[34] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. BigBench: towards an industry standard benchmark for big data analytics. In Ross et al. [55], pages 1197–1208.

[35] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: a database benchmark based on the Facebook social graph. In Ross et al. [55], pages 1185–1196.

[36] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *ICDE Workshops*, pages 41–51. IEEE, 2010.

[37] Joseph E. Hoag and Craig W. Thompson. A parallel general-purpose synthetic data generator. *SIGMOD Record*, 36(1):19–24, 2007.

[38] John M. Stephens and Meikel Poess. MUDD: a multi-dimensional data generator. In Jozo J. Dujmovic, Virgílio A. F. Almeida, and Doug Lea, editors, *WOSP*, pages 104–109. ACM, 2004.

[39] Pengyue J. Lin, Behrokh Samadi, Alan Cipolone, Daniel R. Jeske, Sean Cox, Carlos Rendón, Douglas Holt, and Rui Xiao. Development of a Synthetic Data Set Generator for Building and Testing Information Discovery Systems. In *ITNG*, pages 707–712. IEEE Computer Society, 2006.

[40] Nicolas Bruno and Surajit Chaudhuri. Flexible Database Generators. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB*, pages 1097–1107. ACM, 2005.

[41] David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weber. A framework for testing database applications. In *ISSTA*, pages 147–157, 2000.

[42] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. An agenda for testing relational database applications. *Softw. Test., Verif. Reliab.*, 14(1):17–44, 2004.

[43] Jian Zhang, Chen Xu, and S. C. Cheung. Automatic generation of database instances for white-box testing. In *COMPSAC*, pages 161–165. IEEE Computer Society, 2001.

[44] Arvind Arasu, Raghav Kaushik, and Jian Li. Data generation using declarative constraints. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *SIGMOD Conference*, pages 685–696. ACM, 2011.

[45] Kiran Lakhotia, Mark Harman, and Phil McMinn. A multi-objective approach to search-based test data generation. In Hod Lipson, editor, *GECCO*, pages 1098–1105. ACM, 2007.

[46] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen: generating query-aware test databases. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 341–352. ACM, 2007.

[47] Kenneth Houkjær, Kristian Torp, and Rico Wind. Simple and realistic data generation. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB*, pages 1243–1246. ACM, 2006.

[48] Ray J. Paul, Vlatka Hlupic, and George M. Giaglis. Simulation modelling of business processes. In *Proceedings of the 3 rd U.K. Academy of Information Systems Conference, McGraw-Hill*, pages 311–320. McGraw-Hill, 1998.

[49] Jarg Becker, Martin Kugeler, and Michael Rosemann. *Process Management: a guide for the design of business processes: with 83 figures and 34 tables.* Springer, 2003.

[50] Averill M Law, W David Kelton, and W David Kelton. *Simulation modeling and analysis*, volume 2. McGraw-Hill New York, 1991.

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

[52] Naiqiao Du, Xiaojun Ye, and Jianmin Wang. A semantic-aware data generator for ETL workflows. *Concurrency and Computation: Practice and Experience.*

[53] Il-Yeol Song and Esteban Zimányi, editors. *DOLAP 2009, ACM 12th International Workshop on Data Warehousing and OLAP, Hong Kong, China, November 6, 2009, Proceedings.* ACM, 2009.

[54] Il-Yeol Song and Panos Vassiliadis, editors. *DOLAP 2006, ACM 9th International Workshop on Data Warehousing and OLAP, Arlington, Virginia, USA, November 10, 2006, Proceedings.* ACM, 2006.

[55] Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* ACM, 2013.

# APPENDIX

## A.1 ETL Operation Semantics Definition

| Operation Level | Operation Type | Operation Semantics |
|---|---|---|
| Value | Value Alteration | $\forall(I,O,X,\mathbb{S},A)\ (F(I,O,X,\mathbb{S},A) \rightarrow (SO{=}SI \wedge \lvert O\rvert{=}\lvert I\rvert))$<br>$\forall t_i{\in}I\ (S_1(t_i[X]) \rightarrow \exists t_o{\in}O\ (t_o[SO \setminus A]{=}t_i[SI \setminus A] \wedge t_o(A){=}S_2(t_i[X])))$ |
| Tuple | Replicate Row | $\forall(I,O,X,S,A)\ (F(I,O,X,S,A) \rightarrow (SO{=}SI \wedge \lvert O\rvert{>}\lvert I\rvert))$<br>$\forall t_i{\in}I,\ \exists O'{\subseteq}O\ \lvert O'\rvert{=}n$[1] $\wedge\ \forall t_j{\in}O'\ t_o{=}t_i$ |
| | Router | $\forall(I,\mathbb{O},\mathbb{X},\mathbb{S},A)\ (F(I,\mathbb{O},\mathbb{X},\mathbb{S},A) \rightarrow \forall j(SO_j{=}SI \wedge \lvert O_j\rvert{\leq}\lvert I\rvert))$<br>$\forall j,\ \forall t_i{\in}I\ (S_j(t_i[X_j]) \rightarrow \exists t_o{\in}O_j\ t_o{=}t_i)$ |
| | Join | $\forall(\mathbb{I},O,\mathbb{X},S,A)\ (F(\mathbb{I},O,\mathbb{X},S,A) \rightarrow (SO{=}SI_1 \cup SI_2 \wedge \lvert O_j\rvert \leq \lvert I_1 \times I_2\rvert))$<br>$\forall t_i{\in}I_1,\ \forall t_j{\in}I_2\ S(t_i[X_1],\ t_j[X_2]) \rightarrow \exists t_o{\in}O\ (t_o{=}t_i \bullet t_j)$ |
| | Union | $\forall(\mathbb{I},O,X,S,A)\ (F(\mathbb{I},O,X,S,A) \rightarrow (SI_1{=}SI_2 \wedge SO{=}SI_1 \wedge \lvert O\rvert{=}\lvert I_1\rvert + \lvert I_2\rvert))$<br>$\forall t_i{\in}(I_1 \cup I_2) \rightarrow \exists t_o{\in}O\ (t_o{=}t_i)$ |
| Dataset | Aggregation | $\forall(I,O,X,S,A)\ (F(I,O,X,S,A) \rightarrow (SO{=}X \cup A \wedge \lvert O\rvert{\leq}\lvert I\rvert))$<br>$\forall I'{\in}2^I(\forall t_i{\in}I'\ (\forall t_j{\in}I'\ (t_i[X]{=}t_j[X]) \wedge \forall t_k{\in}I \setminus I'\ t_i[X]{\neq}t_j[X])) \rightarrow \exists!\ t_o{\in}O\ (t_o[X]{=}t_i[X] \wedge t_o[A]{=}S(I'))$ |
| | Sort | $\forall(I,O,X,S,A)\ (F(I,O,X,S,A) \rightarrow (SO{=}SI \wedge \lvert O\rvert{=}\lvert I\rvert))$<br>$\forall t_i{\in}I,\ \exists t_o{\in}O\ (t_o{=}t_i)$<br>$\forall t_o,t_{o'}{\in}O\ (t_o[X]{<}t_{o'}[X] \rightarrow t_o{\prec}t_{o'})$ |
| | Duplicate Removal | $\forall(I,O,X,S,A)\ (F(I,O,X,S,A) \rightarrow (SO{=}SI \wedge \lvert O\rvert{\leq}\lvert I\rvert))$<br>$\forall t_i{\in}I,\ \exists!\ t_o{\in}O\ (t_o{=}t_i)$ |
| Schema | Projection | $\forall(I,O,X,S,A)\ (F(I,O,X,S,A) \rightarrow (SO{=}SI \setminus X \wedge \lvert O\rvert{=}\lvert I\rvert\ ))$<br>$\forall t_i{\in}I,\ \exists t_o{\in}O\ (t_o[SO]{=}t_i[SI \setminus X]))$ |
| | Attribute Addition | $\forall(I,O,X,S,A)\ (F(I,O,X,S,A) \rightarrow (SO{=}SI \cup A \wedge \lvert O\rvert{=}\lvert I\rvert\ ))$<br>$\forall t_i{\in}I,\ \exists t_o{\in}O\ (t_o[SO \setminus A]{=}t_i[SI] \wedge t_o[A]{=}S(t_i[X]))$ |
| Relation | Pivot | $\forall(I,O,X,S,A)\ (F(I,O,X,S,A) \rightarrow (SO{=}(SI \setminus X) \cup A \wedge \lvert O\rvert{=}\lvert I\rvert_a \wedge \lvert I\rvert{=}\lvert O\rvert_a\ ))$<br>$\forall t_i{\in}I,\ \forall a{\in}SI,\ \exists t_o{\in}O,\ \exists b{\in}SO\ (t_o[b]{=}t_i[a]))$ |

Table 1: Table of ETL operations semantics

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